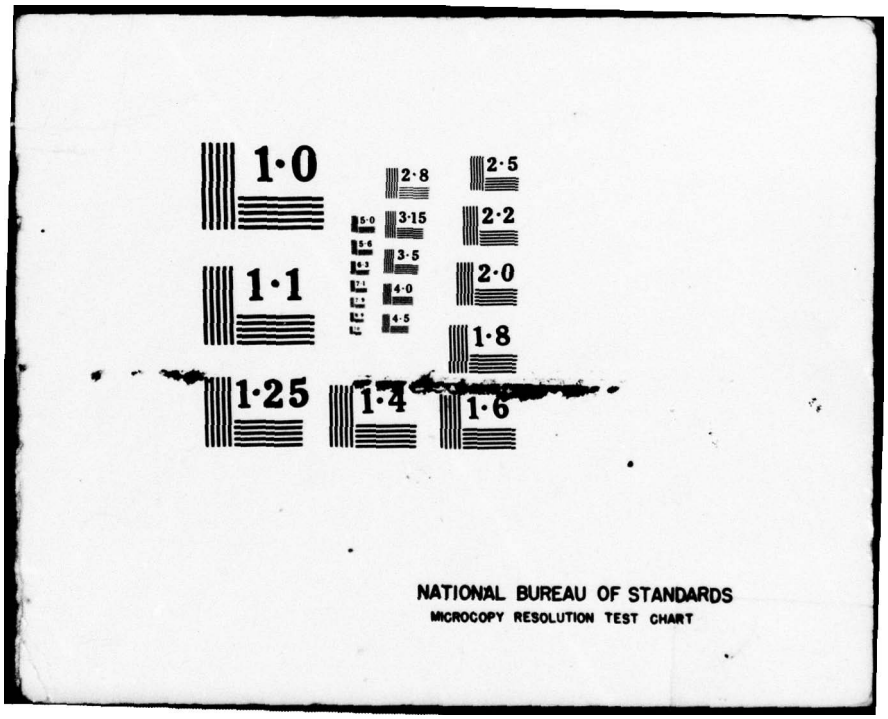
AD-A049 038 TRW DEFENSE AND SPACE SYSTEMS GROUP REDONDO BEACH CALIF F/G 9/2
SMITE REFERENCE MANUAL.(U)
NOV 77 F30602-77-C-0089
UNCLASSIFIED TRW-30417-6002-RU-00 RADC-TR-77-364 NL

1 OF 3
AD
A049038


1·0
1·1
1·25
1·4
1·6
1·8
2·0
2·2
2·5
2·8
3·15
3·5
4·0
4·5
5·0
NATIONAL BUREAU OF STANDARDS
MICROCOPY RESOLUTION TEST CHART

RADC-TR-77-364
Reference Manual
November 1977

SMITE REFERENCE MANUAL

TRW Defense and Space Systems Group

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

This report has been reviewed by the RADC Information Office (OI) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public including foreign nations.

RADC-TR-77-364 has been reviewed and is approved for publication.

APPROVED: Frederick A. Normand

FREDERICK A. NORMAND
Project Engineer

APPROVED: Alan R. Barnum

ALAN R. BARNUM
Assistant Chief
Information Sciences Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (ISCA), Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return this copy. Retain or destroy.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

(19) REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER (18) RADC-TR-77-364	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) (6) SMITE REFERENCE MANUAL •		5. TYPE OF REPORT & PERIOD COVERED Reference Manual
	(14) TRW-30417-6002-RU-00	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) TRW Defense and Space Systems Group	(15) F30602-77-C-0089 new	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS TRW/Defense and Space Systems Group One Space Park Redondo Beach CA 90278	(16) 63728F 55501701	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS (17) 17
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (ISCA) Griffiss AFB NY 13441	(11) November 1977	12. REPORT DATE 13. NUMBER OF PAGES 217
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Same	(12) 224 p.	15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Frederick A. Normand (ISCA)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Hardware Description Language, emulation, simulation, computer architecture

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This is a reference manual for the use of SMITE which is a high level hardware description language used to describe various computer hardware architectures and then to generate an emulation of these architectures on the Nanodata QM-1 microprogrammable computer.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

409637

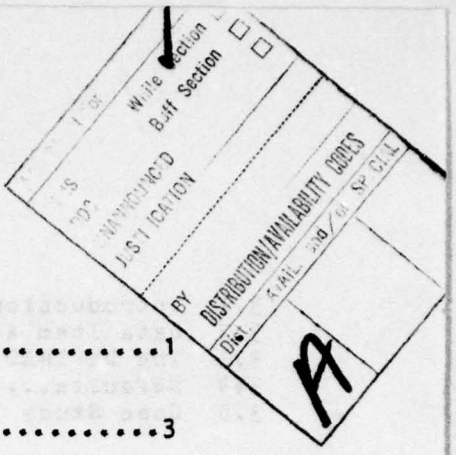


Table of Contents

EVALUATION

RADC is currently building a computer emulation facility to assist in the evaluation of hardware/software/firmware tradeoffs necessary in the development of system architectures. As part of this effort, RADC has purchased a QM-1 microprogrammable computer which is designed for computer emulation. However, the programming of emulations on a microprogrammable computer at the microcode level is a difficult and time consuming task.

The objective of this effort is to obtain SMITE which is a Higher Order Language for describing computer architecture emulations and a compiler which produces code to emulate said architectures on the QM-1 microprogrammable computer.

The SMITE Reference Manual describes the Higher Order Language, how it is used, and how an emulation can be produced.

Frederick A. Normand

FREDERICK A. NORMAND
Project Engineer

Preface

This book is written for the person who wants to understand the use of the SMITE computer description language in the solution of problems of computer architecture and emulation in computer information systems development. The book has three distinct roles:

1. It is the primary material used in formal training classes on SMITE.

2. It is suitable for self-study use by persons familiar with basic computer architecture and emulation concepts.

3. It is suitable for use as a reference manual for users of the SMITE language, compiler, and associated support software.

The work of Daniel D. McCracken in advancing the art of language guide development was invaluable in the writing of this book. In particular, his work "A Guide to FORTRAN IV Programming" served as the example we tried to follow in setting the outline and style we used. Any errors, omissions, or faults, however, are solely our own.

Redondo Beach, California
1977

1. The Fundamentals of SMITE

1.1 The Applications of SMITE

The consideration of computer architecture has been of concern to computer and system designers since the days of Von Neumann and earlier. Our understanding of computer architecture has become considerably more complex than the early partitioning by Von Neumann shown in Figure 1.

Today, we recognize that several levels of interest are crucial to computer architects. These levels are the gate level, the register transfer level, and device level. (Bell and Newell [1] distinguish six levels; although important from the total systems design viewpoint, the other three levels of Bell and Newell are irrelevant to an understanding of SMITE and are therefore not included in this presentation.)

At the highest level, a computer system is the interconnection of devices, such as processors, memories, switches, interfaces, busses, modems, and the like. Commercial computer installations are almost invariably configured at this level. A wide variety of analysis tools have been evolved, of which ECSS [2] and PMSL [3] are representative.

The register transfer level describes a computer system as the interconnection of circuits and components, such as ALUs (arithmetic logical units), registers, data paths, etc. The register transfer level is that hardware representation typically encountered by the assembly language programmer, and is presented in a "hardware reference" or "principles of operation" manual.

The gate level introduces even more detail and complexity into the computer description. A gate level computer description is in terms of AND, OR, and NOT gates, decoders, counters, flip-flops, etc. Languages have been defined for this level, including AHPL [4] and CDL [5]; traditionally this level has been expressed with boolean equations.

Clear distinctions between these three levels do not exist.

For example, memory is found as an entity in both device and register transfer descriptions, and registers are found in both register transfer and gate descriptions. The entire spectrum has become increasingly blurred as circuit integration technology has increased the sophistication of the individual component. Bubble

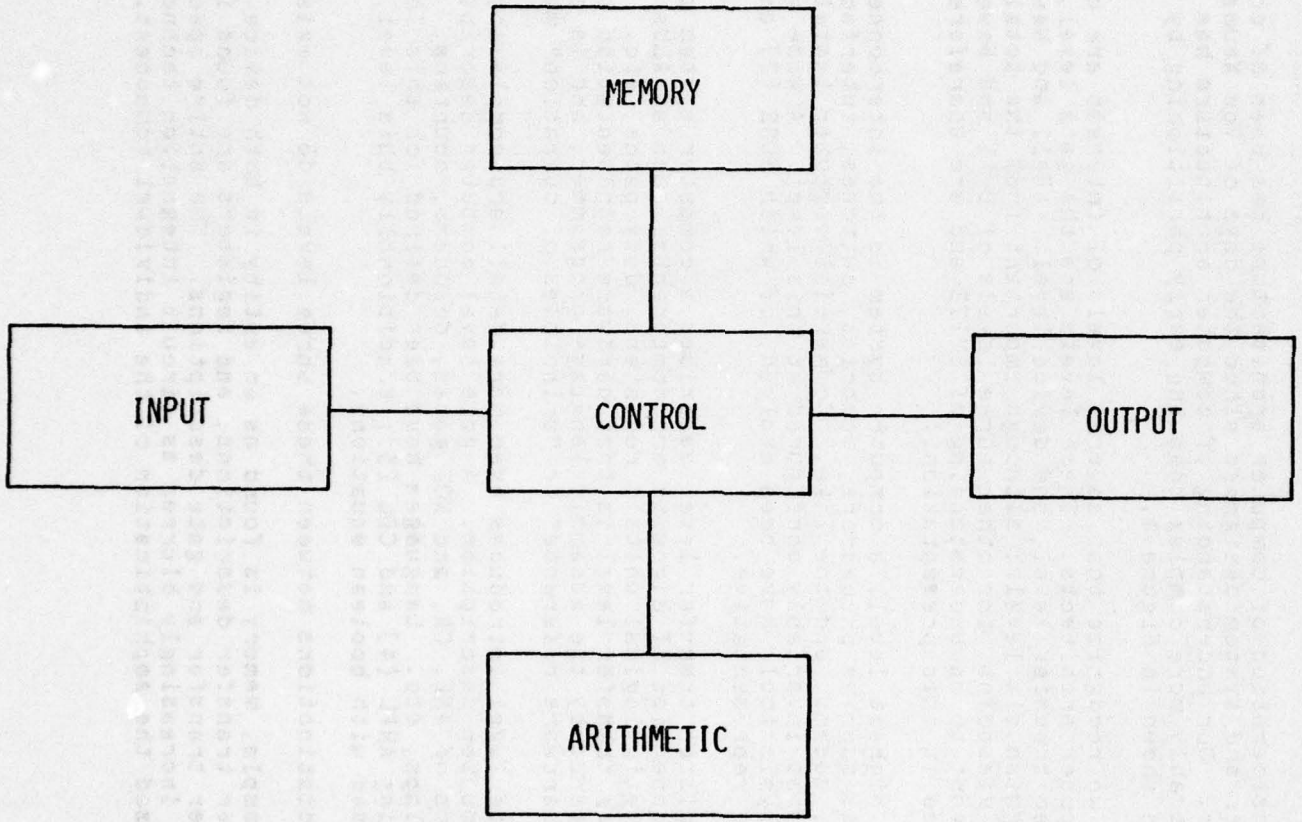


FIGURE 1: CLASSICAL VON NEUMAN CPU ARCHITECTURE
(CIRCA 1942)

memories and complete microprocessors on one or two chips bring complete device level components to the physical scale of the small scale integration flip-flop.

In spite of the difficulty in specifying the exact domain of each description level, the division remains useful. Specification of processor architecture in a formal way, using languages such as SMITE or ISP [6], permits the investigation of performance in a more disciplined, in-depth manner than simple benchmarking, and also permits the development of computer emulations from the formal description [7]. Other efforts to apply register transfer descriptions are also under way, including automatic hardware design, automatic compiler targeting, and program proof.

The SMITE computer description language is intended for analysis of computer architecture and the development of microprogrammed emulations of computer architectures. The context of application of a SMITE-developed computer emulation is diagrammed in Figure 2. (We present only a brief discussion here. Fuller exposition may be found in [7], [8], and [9].)

The complete emulation facility, which we call the Flexible Analysis, Simulation, and Test (FAST) facility, provides a total simulation and analysis environment, including both the emulated hardware and the simulated external environment, as well as test and performance measurement tools. The computers shown in Figure 2 are the Nanodata QM-1 and a large, general purpose computer. Current SMITE software utilizes CDC CYBER 174 equipment as the general purpose host. The Honeywell 6180 or DEC System 20 are possible alternatives.

Along with the SMITE emulators, programs executing in the QM-1 also include target (emulated) machine software, environmental simulations (or interfaces to environmental simulations resident on the general purpose host), performance measurement software, and the QM-1 resident operating system. The general purpose host executes compilers for target software, such as JOVIAL, meta-compilers for development and maintenance of JOVIAL, SMITE, and other compilers, and the SMITE compiler itself.

In Figure 2, the upper half of the diagram (labeled "Software Loop") uses tools to support software development, validation, and verification. The lower half (labeled "Hardware Loop") uses tools for the specification and evaluation of hardware designs. Interaction between the two loops reflecting instruction set changes into the target software is presently supported by compiler modification through a meta-compiler. Future technology development is planned to make this process automatic.

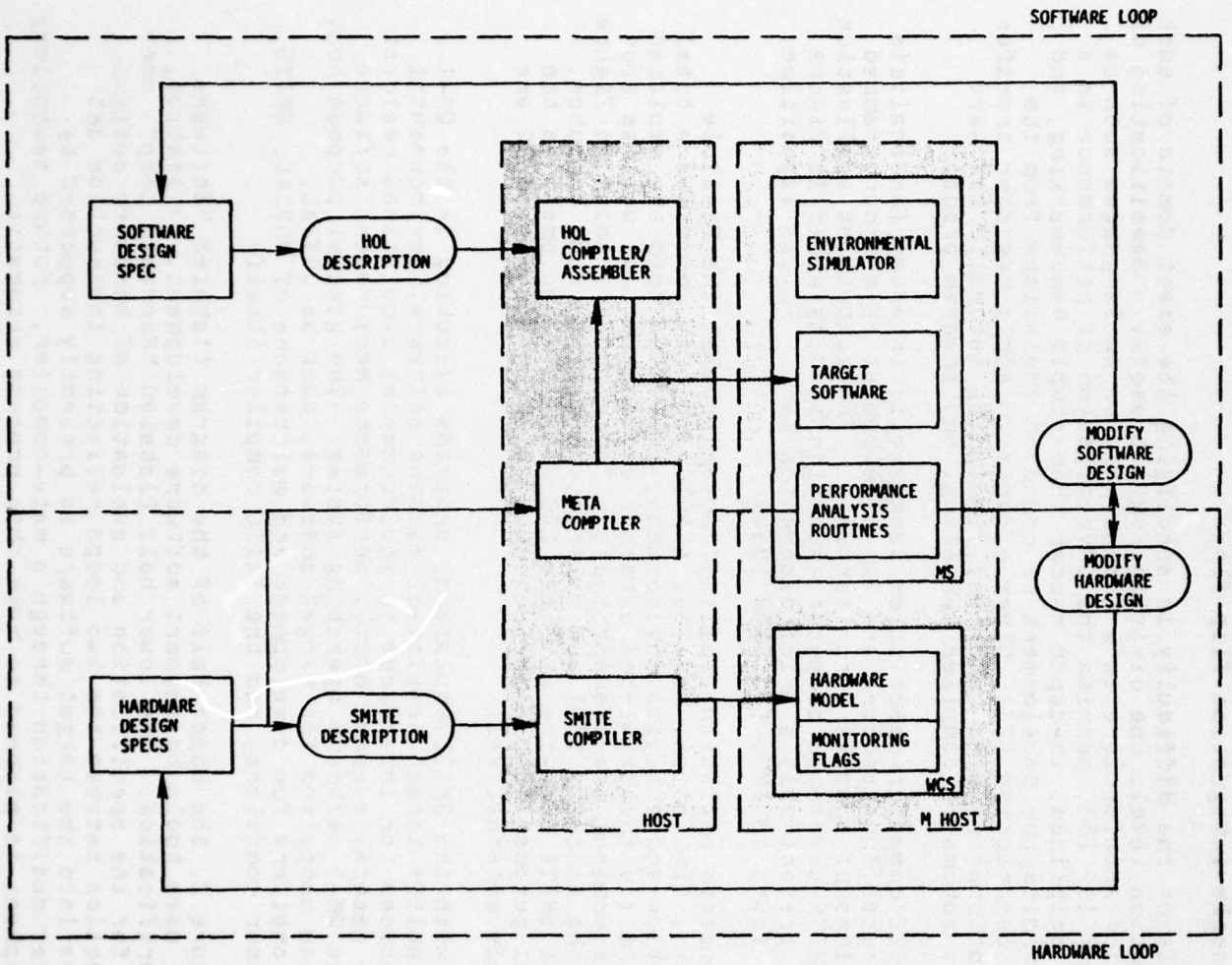


Figure 2 SMITE Emulation Application Context

The role of the SMITE emulator in this context is to provide the capability for executing programs in the target machine instruction set, and for timing the execution of those programs. The QM-1 resident operating system supports emulator execution, and the environmental simulation provides inputs to and processes outputs from the emulation. The SMITE compiler generates emulators from computer descriptions.

This book is about the latter step: the development of SMITE computer descriptions and the processing of those descriptions by the SMITE compiler to form QM-1 microcode emulations. We will at all times discuss SMITE as a problem solving tool. SMITE provides a convenient, expressive notation for describing computer designs at the register transfer level, and a means for evaluating those designs. SMITE is a means for exploring alternatives, not a substitute for invention and creativity on the part of the computer systems architect.

1.2 The Steps in Using SMITE

Using SMITE to help solve architecture problems or to provide a software development testbed involves more steps than simply coding and compiling a description. Instead, using SMITE is a software development process, and done properly entails all the steps required for good software engineering discipline. In this section we describe generically the steps we recommend for the successful use of SMITE.

Problem Formulaton and Requirements Definition

The first step in the SMITE architecture methodology is to specify the problem to be solved using SMITE. Within the total problem of "Will a Motorola 6800 process fire control data fast enough to fuze and launch a salvo of 12 air-to-ground rockets in 2 minutes?," for example, SMITE can provide answers by experiment to questions such as "How long does fuzing a single rocket take?" or "What data rate of input to the fire control system exceeds a limit of 50K instructions per second for input data processing?" In general, this first step entails understanding how the software execution capability and quantitative data available from SMITE emulators contributes to solving the total system problem, and then defining the required tasks and experiments to utilize the emulator.

Once the emulator usage is analyzed, a requirements statement

can be written for development of the emulator. In addition to execution of the computer instruction set, the requirements specification should also specify:

1) The requirements on accuracy and fidelity of timing for instruction interpretation;

2) The functionality of all the external interfaces of the emulator, including input, output, and interrupts;

3) The program load and dump mechanisms, and the operator console interfaces;

4) The data collection and reduction requirements.

An example emulator requirements specification for an emulation of the Raytheon Fault Tolerant Spaceborne Computer (FTSC) is provided as Appendix F. The emulation was intended for software development, integration, and test.

Design Methodology and Structure

The next step in the application of SMITE is the development of the methodology and structure for the design of the emulator. For straightforward emulator development these concepts are well understood and are described in this section. For descriptions intended for architectural evaluation the concepts are not yet generalized; some of the problems and pitfalls will be described.

The initial step in the design sequence is the definition of the data base, including the machine memories, registers, I/O ports, and addressing conventions. The data base for emulator descriptions is primarily resident in the main processor of the description, and is referenced globally by processors nested within the main one.

The next design step is the analysis of the instruction set to determine the decoding that the emulator will perform. In the case of the Intel 8080 microprocessor (Reference the complete description in Appendix C), the primary decode is performed on the leftmost 2 bits of the 8-bit instruction. In some cases a secondary decode is performed on the rightmost three bits, such as to determine the ALU function; certain cases of these secondary decodes further require a third level of decode on the remaining 3 bits. By way of contrast, a description of the CDC-6000 series central processor only requires a single decode on the leftmost 6 bits of the instruction.

Some design freedom exists in the way decode is performed. Returning to the Intel 8080, for example, an alternative to the three-level decode described above is to perform a single 256-way decode on all 8 bits of the instruction. Another alternative is to concatenate the leftmost 2 bits with the rightmost three bits and decode the resulting 5-bit field, using second level decodes as necessary. The consequences of exercising the possible options are to trade execution speed of the emulator against the size of the microcode generated during compilation. The 3-level Intel 8080 decode, for example, will result in a smaller but slower emulation than one performing a 1-level decode.

Once the decode is analyzed, the classes of instructions resulting from decode can be examined to determine the processors (e.g. procedures, functions, or subroutines in other languages) which will be useful in describing the computer. In the Intel 8080, for example, processors to perform an arithmetic operation, to fetch an address from memory, to push or pop the stack, and others were found useful. The external characteristics of these processors, their inputs, outputs, and function, can now be designed. Virtually every emulation will have a processor corresponding to the *ALU (arithmetic-logic-unit), which performs a specified* operation and sets condition bits such as overflow. In the Intel 8080 description (Appendix C), this processor is named PERFORM-ADD.

The final design step is to analyze and specify the design of the external interfaces: Input, output, interrupts, and operator console. The requirements for I/O bear heavily here, as do the facilities available at run time to support the emulation. Conventions and procedures for using the SMITE Application Support Software (SASS) to aid external interface are defined in Chapter 6.

Development Methodology

The emulator may then be coded. The goals of emulator coding should be clarity, readability, and maintainability foremost - individual coding (as opposed to design) practices usually have little impact on emulator performance or size. Some suggested coding practices are listed in Appendix G. In general, good coding practice as applied to top-down, structured programming carries over well into the SMITE domain. Consistency of style is particularly a virtue, so that internal clarity of the emulation and non-astonishment of the reader is obtained. SMITE is well suited for top-down development and integration; in the coding of the FTSC

emulation (Appendix H) for example, the data declarations and instruction decode were first coded and debugged. As individual instruction descriptions and necessary processors were written they were integrated into the growing emulator skeleton.

At some point the external interfaces and emulator-specific support software must be developed also. This software is largely coded in MULTI, the standard QM-1 microinstruction set used by SMITE. The conventions and techniques involved are discussed in chapters 6 and 7. Information about MULTI and the QM-1 may be found in references [10] and [11], respectively. Four new microinstructions have been added to MULTI for SMITE, and two of the standard microinstructions have been changed. These additions and changes to MULTI are defined in Appendix D.

The final step in emulator development is to integrate and test the software. The target computer program used to drive the emulator during final testing is important. If a diagnostic program for the emulated computer is available, it is the logical choice. The use of a hardware diagnostic program may be difficult, however, if the program executes illegal opcodes assuming a particular response from the hardware, or otherwise depends on particular construction of the hardware not modeled by the emulation.

Architectural Analysis Methodology

The course of events to follow is less clear for architectural analysis. If the method of analysis is timing the execution of benchmarks or complete software systems on an emulator, as might be done to evaluate the effect of replacing core memory with faster semiconductor memory, a basic emulator including modeling of the instruction timings would be required, and could be developed as outlined above. At the other extreme, if an architecture description is wanted as input to a static analysis to determine the effect of changing the number of ALUs in the implementation, an analysis which is the subject of current research, the methodology for developing the appropriate description is unknown. The problems in developing a methodology begin with requirements. The requirements on a description to be used for static analysis are not completely known. For example, the SMITE code to describe the INTEL 8080 add (reference Appendix C), is

```
(AUX-CARRY//A<3:0>) <- CARRY-IN + A<3:0> + OP-REG<3:0>;
CARRY-A <- AUX-CARRY + A<7:4> + OP-REG<7:4>;
SIGN <- A<7>;
```

```
ZERO <-  A = 0;
PARITY <- A<7> XOR A<6> XOR A<5> XOR A<4>
   XOR A<3> XOR A<2> XOR A<1> XOR 1;
```

This code is conceivably implemented with two cycles of a 4-bit ALU plus some combinatorial logic, with an 8-bit ALU and combinatorial logic, or with a special purpose device to implement the entire function. The descriptive problem is to describe the structure and operation of the architecture so as to permit identificaton of possible alternatives, and to highlight the opportunities inherent in the architecture design. To date, one principle has been proposed as key to the methodology: All operations within any given statement should be of the same level of complexity, as should all statements in any given sequence. It is still an open research topic, however, how to measure and compare complexity, and how to construct descriptions having this property.

The application of this "leveling" property to the short Intel 8080 addition example above helps to identify some problems. For example,

```
(AUX-CARRY//A<3:0>) <- CARRY-IN + A<3:0> + OP-REG<3:0>;
```

and

```
SIGN <- A<7>;
```

are intuitively not at the same level of complexity. Even

```
(AUX-CARRY//A<3:0>) <- CARRY-IN + A<3:0> + OP-REG<3:0>;
```

and

```
CARRY-A <- AUX-CARRY + A<7:4> + OP-REG<7:4>;
```

are of differing levels of complexity, since in the former statement the concatenation of AUX-CARRY and A<3:0> is described explicitly, while in the latter the concatenation is implicit by reference to CARRY-A, which is defined elsewhere in the description as the concatenation of the CARRY bit with the A register.

1.3 A SMITE Computer Description

An example of a complete architecture is shown in Figure 3. In this example the data processing system under consideration inputs high-data-rate radar returns, performs tracking, identification and discrimination, and outputs airspace data to user consoles. Within the data processing system the architecture to be analyzed consists of a custom array processor for return correlation and track assembly, a Digital Equipment Corporation PDP 11/45 processor equipped with floating point hardware used for system control and mass data base management, and a dual-processor Honeywell H-6180 for the bulk of the computational workload analyzing tracks assembled into the data base. The processors in the system are connected together and to a high bandwidth mass store by high bandwidth busses.

A description of this example would be hierarchical. At the highest level would be the declaration of the three computers and the system mass storage device as entities, and the definitions of the capacity, protocol, and endpoints for each of the bussed interconnections. This level of description is the PMS level discussed earlier, and is a higher level than the register transfer level of a SMITE computer description.

The next hierarchical level down in the system description expands the description of each system entity. The array processor and the dual-processor H-6180 would be described in SMITE. These SMITE descriptions extend out in scope to the boundaries of the processor: everything inside the I/O and interrupt ports that the outside world connects to is described. Connecting the outside world to the emulation requires techniques discussed in chapters 6 and 7.

The PDP-11 however, presents some options. Either the PDP-11 itself can be described as a bus-connected system using PMS concepts, in which case individual SMITE descriptions are required for the CPU, I/O, and memory, or else it can be described as a single computer entirely in SMITE. The latter approach has the disadvantages that changes to the memory or I/O addressing sequences a re-write of the CPU description to reflect the new addressing conventions, and that the bus-oriented nature of the computer is completely obscured from static architectural analysis.

The monolithic approach has the advantage, however, that the description is self-contained and does not require external microcode in the implementation of the CPU.

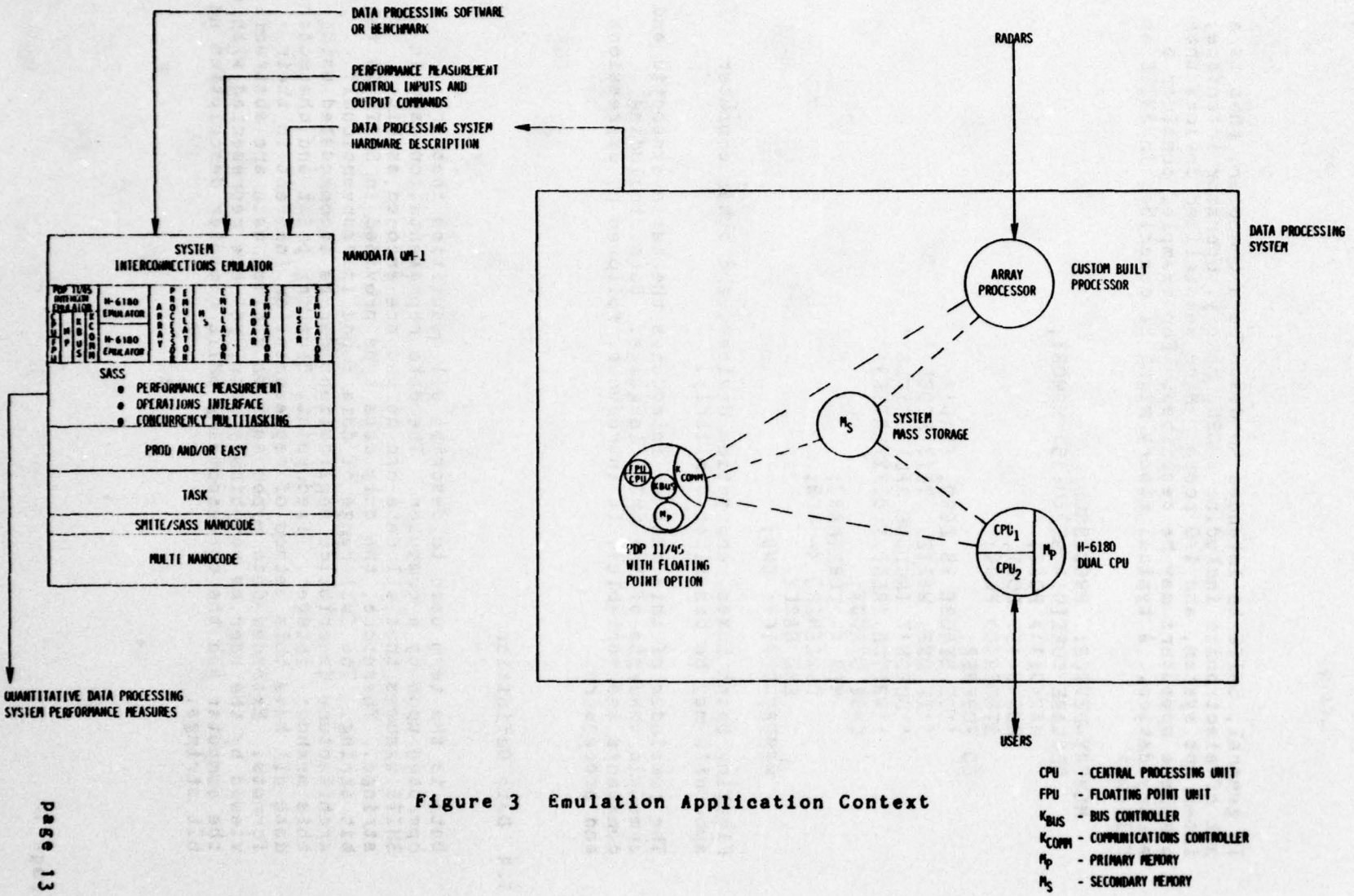


Figure 3 Emulation Application Context

In general, SMITE is intended to describe a computer, that is a set of electronics including a CPU, memory, operator interface, interrupt system, and I/O ports. More specialized devices than complete computers may be described. For example, consider a memory device. A typical memory might be described in SMITE as follows:

```
MEMORY-DEVICE:  PROCESSOR;
   DECLARE CORE[0:4095]<0:15> MEMORY,
      MAR<0:11> PORT,
      MDR<0:15> PORT,
      STROBE<0> PORT;
   DO FOREVER;
      ''IF STROBE IS ZERO, READ''
      ''IF ONE, WRITE.  THIS MODEL''
      ''DOESN'T INCLUDE SPLIT-CYCLE''
      ''WRITES (READ/MODIFY/WRITE)''
      CASE STROBE;
         MDR <- CORE[MAR];
         CORE[MAR] <- MDR;
         END CASE;
   MEMORY-DEVICE: END;
```

Floating point boxes, encryption devices, and other computer sub-units may be described similarly.

The remainder of this chapter introduces the basic syntactic and semantic concepts of the SMITE language. Data, including constants and variables, is introduced, followed by expressions and processors.

1.4 Data Definition

Data is the term used to describe all quantities that are operated upon by a computer. The data representation used in SMITE assumes that all data can be and are encoded as bit strings. Therefore, the only data type provided in SMITE is the bit string. The full range of data found in "conventional" architecture description applications can be accommodated using this method: integer, fixed point, floating point and character data all have this method of representation inherent in their formats. Extended data types such as decimal data are abstractly viewed by the user as non-binary data, yet are represented within the computer and the corresponding SMITE computer description as bit strings.

This section presents the basic concepts of data employed in the SMITE language. The discussion begins with the various forms of constants recognized in SMITE, and then proceeds to the names of data items. The types of data items are discussed next, followed by the statement for declaring data items such a registers, memories, I/O ports, and the like.

1.4.1 Constants

All constants or numbers are assumed to be decimal in SMITE unless indicated otherwise. Binary constants are represented as B'binary string', octal constants as O'octal string', and hexadecimal constants as X'hexadecimal string.

The following are valid SMITE constants:

```
125           (a decimal constant)

X'AB12'       (a hexadecimal constant)

B'1101100'    (a binary constant)

O'7734'       (an octal constant)
```

while the following are invalid SMITE constants:

```
A14A          (A is not a valid decimal digit)

X'Q12'        (Q is not a valid hexadecimal digit)

B'411001'     (4 is not a valid binary digit)
```

All data in SMITE have the characteristic of width, i.e. the number of bits being used to represent a datum is its width in that particular context. The width of a constant is the minimum number of bits needed to represent the number in binary form. The number zero is defined to have a width of one bit. The width of a constant may be increased in expression evaluation (section 1.5). All constants are positive in SMITE. For example,

```
5
```

is a constant with a width of 3 bits, and

```
-5
```

is an expression which computes the complement of the number five. The width of this expression is 4 bits to allow space

for the required sign bit. Some other examples will illustrate the computation of the width of constants:

```
B'11111'
```

has width 5,

```
X'FF'
```

has width 8,

```
O'7046'
```

has width 12,

```
B'00000000001'
```

has width 1, as does

```
X'0001'
```

and

```
O'0001'
```

1.4.2 Storage Organization: The Word

The central element of storage organization in SMITE is the word. A word is any group of one or more bits that is logically referred to as a single entity. The SMITE concept of word therefore encompasses a range of elements commonly found in modern computer structures, from the bit and byte through the halfword and word, to the doubleword. For example, in the IBM System/370, bits, 8-bit bytes, 16-bit halfwords, 32-bit words, and 64-bit doublewords are all supported data types. This word structure could be represented in SMITE as

```
DECLARE
   DOUBLEWORD<0:63>,
      WORD[0:1]<0:31> DEFINED DOUBLEWORD,
         HALFWORD[0:3]<0:15> DEFINED WORD,
            BYTE[0:7]<0:7> DEFINED HALFWORD,
               BITS[0:63]<0> DEFINED BYTE;
```

The storage structure thus represented is shown in figure 4.

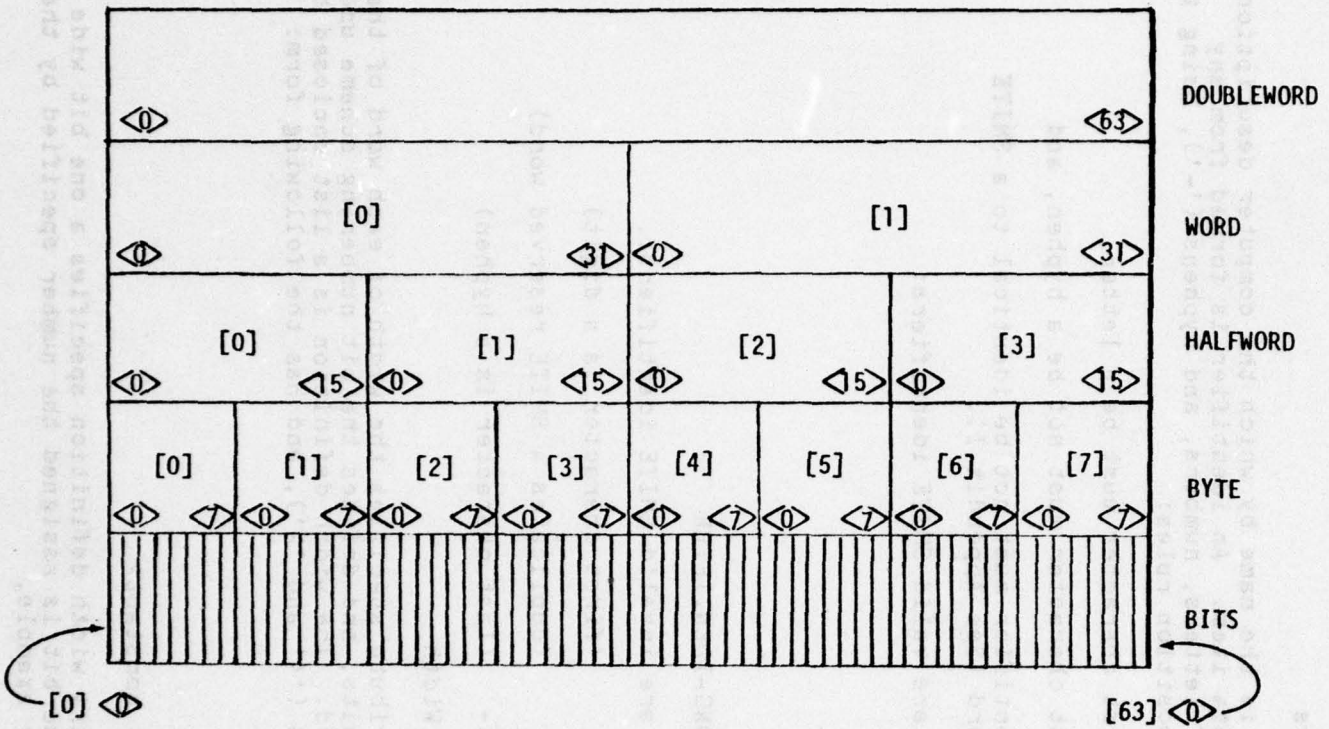


Figure 4. An Example Storage Structure

1.4.3 Identifiers

An identifier is the name by which the computer description refers to a data item. An identifier is formed from any combination of letters, numbers, and hyphens ('-'), using the following composition rules:

1. The first character must be a letter,

2. The last character must not be a hyphen, and

3. The identifier must not be identical to a SMITE reserved word (See Appendix I).

The following are valid SMITE identifiers:

ZERO-FLAG

JABBERWOCKY

R2-D2

A-FAIRLY-LONG-IDENTIFIER

The following are invalid SMITE identifiers:

0-FLAG	(first character is a digit)
DECLARE	(duplicates a SMITE reserved word)
TEST-STATUS-	(last character is a hyphen)

1.4.4 Data Item Width

The width attribute specifies the width of each word of the data item in bits, and defines the bit numbering scheme used within the word. The width definition is a list enclosed by angle brackets ('<' and '>'), and has the following form:

<constant>

or

<constant : constant>

The single entry width definition specifies a one bit wide data item. The bit is assigned the number specified by the constant. For example,

```
DECLARE CARRY-STATUS<7> REGISTER;
```

declares a one bit wide register named CARRY-STATUS. The bit number is (perhaps arbitrarily) assigned the ordinal 7.

The two-entry width definition specifies the width of a multiple-bit word. The leftmost constant in the width definition corresponds to the number of the leftmost bit, and the rightmost constant corresponds to the number of the rightmost bit. For example,

```
DECLARE ACCUMULATOR<7:0> REGISTER;
```

declares an 8-bit register named ACCUMULATOR, with the leftmost bit numbered 7, and the rightmost bit numbered 0. Similarly,

```
DECLARE STATUS-WORD<0:15> REGISTER;
```

declares a 16-bit register named STATUS-WORD, with the leftmost bit numbered 0, and the rightmost bit numbered 15. The bit numbers are not required to origin at zero at either end of the word. Thus,

```
DECLARE LARGE-BIT-NUMBERS<X'FFE':X'FFF'> REGISTER;
```

declares a two bit register.

1.4.5 Storage Arrays

Many storage devices consist of an ordered set of words with an addressing mechanism to select a particular word from the entire set. SMITE provides an array definition facility to describe such structures. Arrays are sequentially ordered sets of words, each of the same width, accessed by an address. The address space is defined for an array by specification of the lower and upper address bounds.

Length definition attributes are used to describe array data items. The definition consists of two constants separated by a colon and enclosed by square brackets, as follows:

```
[constant : constant]
```

The length attribute defines a block of words (the block length is defined to be one if the two address bounds are equal), with the leftmost entry as the index of the first word in the block and the rightmost entry as the index of the last

word in the block. The first word address is constrained to be less than or equal to the last word address. The statement

```
DECLARE TO-BE[1:20];
```

declares a contiguous block of 20 words, with addresses ranging consecutively from 1 to 20, inclusive. Similarly, the statement

```
DECLARE NOT-TO-BE[0:X'FF'];
```

declares an array of 64 words with consecutive addresses from 0 to 63, inclusive.

The declaration of an array implies the existence of address decode and selection logic. Thus, there is a subtle difference between the declaration

```
DECLARE OP-HOLD[0:0];
```

and the declaration

```
DECLARE OP-HOLD;
```

The first declaration defines an array one word long, including the (unnecessary) address selection logic, while the second defines a word. The former statement is liable to confuse the reader of a computer description, and should only be written if such hardware actually exists.

1.4.6 Storage Classes

Several different types of storage elements are generally found in computers, such as registers, memories, input/output ports, and others. SMITE supports the declaration of different types of storage elements with the storage class attribute, which is supplied as part of the declaration for a data item. Storage class attributes are discussed further in chapter 3. Two of the storage class attributes available in SMITE are for declaring registers and memories. For example,

```
DECLARE ACCUMULATOR<0:35> REGISTER;
```

defines a 36-bit register, and

```
DECLARE MEM[0:4095]<0:15> MEMORY;
```

declares a 16-bit by 4096-word memory as might be found in a PDP-11.

1.4.7 Data Item Declaration

Every data item used in a SMITE computer description must be declared before it may be used. Data items are defined in SMITE by the use of the DECLARE statement, which has the following form:

```
DECLARE identifier attribute(s);
```

The DECLARE statement also allows the declaration of more than one data item. Individual item declarations in a single statement are separated by commas. For example,

```
DECLARE

   identifier attribute(s),

   identifier attribute(s);
```

Within a processor, any number of DECLARE statements may appear, however they must all be grouped at the beginning. The declarations define all registers, memories, and other data definitions needed for the description.

Each individual data item declaration within the DECLARE statement has the following form:

```
identifier length-attribute width-attribute class-attribute
```

The length attribute need only be present if an array is being declared.

An additional attribute, DEFINED, is presented in chapter 3. The following are examples of valid SMITE data item declarations:

```
DECLARE
   STATUS-WORD<3:8> REGISTER;
DECLARE
   ACCUMULATOR<0:7> REGISTER,
   INDEX<0:15> REGISTER,
   MEM[0:0'7777']<0:15> MEMORY;
```

1.4.8 Data Item References

References to storage elements are similar in form to the storage element declarations, but are interpreted differently. The possible forms for a data item reference are

```
   identifier

   identifier word-selector

   identifier subfield-selector
```

or

```
   identifier word-selector subfield-selector
```

A word selector looks like a length attribute, and is of the form

```
   [ expression ]
```

Expressions are discussed in section 1.5; essentially, an expression is any computable value (e.g. constants, data items, operations on data items, processor function calls, etc.). The expression in a word selector acts as a subscript, and is input to the address selection logic for the array to determine the specific word to be referenced. Word selectors may be included in a data item reference only if the data item was defined as an array when it was declared, ad may not be omitted if the item being referenced is an array. Given the declaration

```
   DECLARE
      ARRAY[0:15]<0:63> MEMORY,
      MAR<0:3> REGISTER;
```

then

```
   ARRAY[0]
```

references the first word of the array, and

```
   ARRAY[MAR]
```

references the word of the array addressed by the value of data item MAR.

A subfield selector looks like a width attribute, and is of the form

```
   < constant >
```

or

```
   <constant : constant>
```

If the width definition is omitted the entire word is referenced; otherwise the subfield identified by the width definition is referenced. Given the declaration

```
DECLARE ACCUM<7:0> REGISTER;
```

then

```
ACCUM
```

references the entire eight bits, while

```
ACCUM<5:4>
```

references a two bit subfield of ACCUM.

The word selector and width selector may be used in conjunction for array references. Using the earlier example given for word selectors,

```
DECLARE
   ARRAY[0:15]<0:63> MEMORY,
   MAR<0:3> REGISTER;
```

then

```
ARRAY[MAR]<4>
```

references a one bit subfield of the word addressed by MAR, and

```
ARRAY[1]<2:3>
```

references a two bit subfield of the second word of the array.

1.4.9 Limits

Certain limits on data item length and width, and on word selector expression width, are imposed on the SMITE language by the SMITE compiler and the Nanodata QM-1 computer. These limits are as follows:

1. No data item may have a word width of greater than 72 bits. In the case of array declarations, this means that the width of each individual word of the array has a maximum of 72 bits. In addition, no microinstructions have been implemented on the QM-1 to shift quantities greater than 36 bits in width, and therefore descriptions

incorporating data items of from 37 to 72 bits may not be executable.

2. The total main store allocation for an emulator is limited to an 18-bit address by the architecture of the QM-1, and may be further limited by the amount of main store installed on a particular QM-1. Thus while there is no specific limit on the number of data items that may be declared, or on the array length of any one data item, the maximum available main store address space may not be exceeded by the aggregate data item declarations.

3. Due the means of storage allocation adopted in the SMITE compiler, there is a maximum width for any word selector. For reference to data items of from 1 to 36 bits in width, this maximum is 17 bits. For reference to data items of from 37 to 72 bits, this word selector maximum width is 16 bits.

4. The SMITE compiler does not process constants of width greater than 60 bits.

1.5 Operations and Expressions

All data manipulation (i.e. moving and/or operating on data) is achieved in SMITE by the use of expressions. Processor invocation is also accomplished through expressions. An expression is a combination of data item references, constants, operators and parentheses producing a single value upon evaluation.

An expression is one type of SMITE executable statement. The expression statement has the form:

```
expression ;
```

The binary operators defined in SMITE are:

arithmetic (two's complement)

+	addition
-	subtraction

relational (inequality comparisons are two's complement. The

relationals are defined for both signed and unsigned quantities)

=	equal to
/=	not equal to
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to

boolean

OR	logical sum
AND	logical product
XOR	logical difference

miscellaneous

<-	data transfer
//	concatenation

The unary operators defined in SMITE are:

+	(unary plus is ignored)
-	arithmetic negation (two's complement)
NOT	logical negation (one's complement)

Multiple character operators must be coded without spaces between the characters forming the operator.

Expressions are composed of one or more terms (operands) separated by binary operators. A term can be a storage element reference, processor call, constant, an expression surrounded by parentheses, or a unary operator with any of the previous four items as the operand. The following are expressions:

```
AFB
(A + C) // D
```

```
A // B + D = Q

A + B > (Q + C) OR S

A <- B + (-C + 5) // D <- E<3:4>
```

Expressions are evaluated from right to left, with the exception that unary operators are evaluated before binary operators, and the special binary operator '//' (concatenation) is evaluated even before unary operators since it is a part of basic operand formation. The normal order of evaluation may be altered by parenthesization. The right-to-left convention, although a departure from most procedural languages, has the benefit that all evaluation and operator definition has is a simple rule to define the order of computation, rather than the complicated tables of operator precedence found in many other languages. In addition, the evaluation from right to left makes the embedding of transfer operators in an expression natural. (i.e. B <- C + D <- MEMORY[A] is interpreted as: fetch MEMORY[A], store that value into D, add the same value to C and store the result in B).

The operands allowed on the left hand side of the transfer operator ('<-') are restricted to storage elements and the names of function processors. For example,

```
A <- 4
```

and

```
A//B <- C
```

are a valid assignments, but

```
4 <- A
```

and

```
(C + D) <- E
```

are not.

Explicit rules have been defined in SMITE for the width of a result after the evaluation of an expression. These rules are as follows:

1. The width of a data item reference is the declared width of the data item unless a subfield extraction has been included in the data item reference, in which case the width of the reference is established by the subfield reference.

The width of a constant is the minimum number of bits required to hold the constant.

2. The width of an expression after the evaluation of the unary plus and NOT operators is the same as the width of the operand. The width of an expression after the evaluation of a unary minus operator is also the same as the width of the operand unless the operand is a constant, in which case the width of the expression is one greater than the width of the constant.

3. The width of an expression after the evaluation of any binary arithmetic or logical operator is the same as the width of the two operands if these widths are identical. If the widths of the two operands are not equal, then the smaller-width operand is expanded to the width of the wider operand, and the operation performed. The width of the result will then be the width of the wider operand. When the smaller operand is expanded, it will be zero-padded on the left unless the sign extension (SE) function is used (see section 1.5).

4. The width of an expression after evaluation of any relational operator is one bit. The operands are adjusted to conforming widths as in rule 3 above before comparison if their widths are not equal. The relational operation is performed assuming both operands are unsigned magnitudes unless the sign extension (SE) function is used on one of the operands. In the latter case, an arithmetic comparison assuming both operands are signed is performed.

5. The width of an expression after evaluation of a concatenation operator is the sum of the widths of the two operands.

6. If a transfer of data is being performed, the source (right hand) operand is constrained to be no wider than the receiving (left hand) operand. If the source operand is smaller than the destination operand, padding is performed as in rule 3 above. The width of the expression is equal to the width of the destination operand.

1.6 Processors

A SMITE computer description conceivably could be written as one self contained program. However, the description is much more readable if it is separated into smaller modules. In SMITE,

these modules are called processors. The SMITE processor is analogous to the function, subroutine, or procedure capability found in conventional higher order procedural languages. The SMITE processor is typically used to isolate common processing steps and parameter dependent processes, or to modularize the description in order to highlight features of the computer architecture.

SMITE processors may be called re-entrantly, as may occur from within parallel contexts, but may not be used recursively.

1.6.1 Processor Structure

The basic module of a SMITE computer description is the processor. The processor is begun with a processor header, and is terminated by an END statement. The processor header and END statement must both be labeled with an identifier, the name of the processor, and the name must be the same on both statements. The processor returns to the point from which it was called when control passes to the END statement. A processor, in its most basic form, has the framework:

```
identifier: PROCESSOR;
   data declarations  (if any)
   executable SMITE statements  (at least one required)
   identifier: END;
```

For example, the processor to fetch an address from memory in the Intel 8080 description is:

```
GETADD: PROCESSOR;
   OP-PAIR <- MEM[PC+1] // MEM[PC];
   PC <- PC + 2;
   GETADD: END;
```

This example contains no data declaration statements. An example of a simple processor containing data declarations as well as executable statements is the memory processor defined earlier:

```
MEMORY-DEVICE:  PROCESSOR;
   DECLARE CORE[0:4095]<0:15> MEMORY,
      MAR<0:11> PORT,
      MDR<0:15> PORT,
      STROBE<0> PORT;
   DO FOREVER;
      ''IF STROBE IS ZERO, READ''
      ''IF ONE, WRITE.  THIS MODEL''
      ''DOESN'T INCLUDE SPLIT-CYCLE''
```

```
      ''WRITES (READ/MODIFY/WRITE)''
      CASE STROBE;
         MDR <- CORE[MAR];
         CORE[MAR] <- MDR;
         END CASE;
   MEMORY-DEVICE: END;
```

In addition to data declarations and executable statements, processors may contain definitions of subprocessors. Subprocessor definitions are placed between the last data declaration and the first executable statement of the surrounding processor. Subprocessor definitions themselves have the exact same form as the processor definition just introduced. For example, the memory device example could be re-coded as follows to use subprocessors:

```
MEMORY-DEVICE:  PROCESSOR;
   DECLARE CORE[0:4095]<0:15> MEMORY,
      MAR<0:11> PORT,
      MDR<0:15> PORT,
      STROBE<0> PORT;
   READ: PROCESSOR;
      MDR <- CORE[MAR];
      READ: END;
   WRITE: PROCESSOR;
      CORE[MAR] <- MDR;
      WRITE: END;
   DO FOREVER;
      ''IF STROBE IS ZERO, READ''
      ''IF ONE, WRITE.  THIS MODEL''
      ''DOESN'T INCLUDE SPLIT-CYCLE''
      ''WRITES (READ/MODIFY/WRITE)''
      CASE STROBE;
         READ;
         WRITE;
         END CASE;
   MEMORY-DEVICE: END;
```

Subprocessors may themselves, in turn, contain other subprocessors. Data declared within a subprocessor is temporary data which is lost after the processor is exited.

1.6.2 Scope of Names Recognition

The last example illustrates the use of global data by subprocessors. In that example, the ports MAR and MDR, and the array CORE, were accessed globally by the subprocessors. This usage is an example of the general concept in SMITE of scope of name recognition. SMITE allows data items to be

referenced globally from within subprocessors, and also allows subprocessors to redefine data item and processor names. The rules for scope of names recognition are as follows:

1. Within and inclusive of the main processor (the outermost processor of a SMITE computer description), the names known are only those data items and subprocessors declared in the main processor. These names are local to the main processor.

2. Within and inclusive of a subprocessor (whether the subprocessor was itself declared in the main processor or another subprocessor), data items and processors declared within the subprocessor are known as local names. All processors and data items declared in processors surrounding this subprocessor are known as global names.

3. Any globally known name may be re-declared locally, superseding the global definition.

For example, consider the following skeleton of a SMITE computer description.

```
A: PROCESSOR;
   DECLARE
      U<0:1> REGISTER,
      V<0> MEMORY;
   B: PROCESSOR;
      DECLARE
         W<0:15> REGISTER,
         U<0:15> MEMORY;
      C: PROCESSOR;
         DECLARE
            V <0:15> REGISTER;
         C: END;
      B: END;
   D: PROCESSOR;
      DECLARE
         X<0:15> MEMORY;
      D: END;
   A: END;
```

The names known within the text of processor A are U (a 2-bit register), V (a 1-bit memory, B (a processor), and D (a processor). The names W, C, and X are not known to A. The names known within the text of processor B are V (a 1-bit memory known globally from the definition in A), W (a 16-bit register declared locally in B), U (a 16-bit memory declared locally in B redefining the definition of U in processor A), C

(a processor declared in B), and D (a processor known from the global definition in A). The names known within the text of processor C include everything known in B, plus a redefinition of the global name V (now a 16-bit register). The names known within the text of D include the names declared in A (i.e. U as a 2-bit register, V as a 1-bit memory, and B as a processor), plus X (a 16-bit memory).

1.6.3 Function Processors

In the preceding examples, processors have been used exclusively as subroutines, that is they have been defined and called strictly for their effect, and have had no parameters. SMITE provides the capability for function processors as well as subroutine processors, and permits both types of processors to be defined with parameters. The type and parameters of a processor are specified in the processor header statement.

```
name: PROCESSOR;
```

The header for a simple subroutine processor with no parameters, as used in the previous examples, has the form

If a value (a word) is to be returned as a function result from invocation of the processor, then a width attribute is added to the processor header:

```
name: PROCESSOR width-attribute;
```

For example, a processor to compute an arithmetic/logical function of 8 bits from global data (so no parameters are required) might be defined with the header:

```
ALU: PROCESSOR<0:7>;
```

Processors are invoked in expressions by referencing the processor name. For example, the ALU processor defined above could be invoked and the returned value stored by the statement

```
ACCUMULATOR <- ALU;
```

The ALU processor could also be invoked without the function result being used, such as might be done if only the setting of status bits was desired. A statement to perform that task would be

```
ALU;
```

1.6.4 Processor Parameters

Both subroutine processors and function processors may be defined to receive or return parameters. A subroutine processor header including parameters has the form:

```
name: PROCESSOR ( argument-list );
```

A function processor header including parameters has the following form:

```
name: PROCESSOR width-attribute ( argument-list );
```

The argument list is a list of local data items (formal parameters) which are replaced by the calling (actual) parameters at the time of processor invocation. The number of actual parameters must agree with the number of entries in the argument list.

Processor parameters are passed to and from the processor using either call-by-value (CBV), if the parameter is only referenced in the processor and never assigned a value, or call-by-value-returned (CBR), for parameters which are assigned a value within the scope of the processor. The operation of these parameter passing techniques closely models the operation of many hardware units: the values input to the hardware are gated off of a bus (call-by-value), and the unit performs its function. The returned results, if any, are then gated back onto a bus for transmission to the caller (call-by-value-returned).

For CBV parameters, the actual parameter may be any expression. For CBR parameters, the actual parameter may be any expression allowed on the left hand side of an assignment operator (e.g. data items, concatenated data items, function processor names (for defining the function value), etc.).

If a particular SMITE variable is a calling parameter, and is referenced as a global variable in the processor, the global references and the parametric references use separate data values. For example:

```
A: PROCESSOR;
   DECLARE
      ACTUAL<0:15> REGISTER;
   B: PROCESSOR(FORMAL);
      DECLARE
         FORMAL<0:15> REGISTER;
      FORMAL <- 1;
```

```
      IF FORMAL /= ACTUAL
         THEN ...
         END IF;
      B: END;
   ACTUAL <- 0;
   B(ACTUAL);
   IF ACTUAL = 1
      THEN ...
      END IF;
   A: END;
```

Both IF statements will evaluate the relational test as true. The operation of the example is as follows. The main processor (A) sets ACTUAL to 0. The processor B is then invoked, and the value 0 is bound to FORMAL. B then assigns the value 1 to FORMAL, but that value will not be gated back into ACTUAL until B terminates execution and returns to the point of the call. The IF test in B will therefore find that ACTUAL is in fact not equal to FORMAL. B eventually returns to A, at which time the value assigned to FORMAL within B is gated into ACTUAL. The IF test in A therefore finds ACTUAL to have a value of 1.

Every formal parameter in the processor header must be declared in the subprocessor. If the actual parameter is a simple data item or *a subfield of a simple* data item, then the storage class attribute of the two data items (formal and actual) must be identical (MEMORY and REGISTER are considered identical for this test). The width of the formal parameter must in all cases be greater than or equal to the width of the actual parameter; if the subprocessor assigns a value to the formal parameter, then the widths if the formal and actual parameters must be identical. Parameters may not have a storage class of EXTERNAL, DATA, or PORT, and may not be declared as arrays in the subprocessor.

1.6.5 Intrinsic Processors

SMITE has the following built-in functions to perform shifts and sign extensions:

Function	Operation
SE(D)	Sign extend D
SLL(D,S)	Shift D left logical S bits
SRL(D,S)	Shift D right logical S bits
SLA(D,S)	Shift D left arithmetic S bits
SRA(D,S)	Shift D right arithmetic S bits
SLC(D,S)	Shift D left circular S bits

SRC(D,S) Shift D right circular S bits

In the above definitions, both S and D may be any arbitrary expressions.

The sign extension operator is useful when the width of an operand is to be expanded: the most significant bit of the data word is extended until a data word of the appropriate width is formed. The sign extension operator is also used to indicate that an operand of a relational operator is signed, and therefore that an arithmetic comparison should be made.

The shift operators have no predefined width. The width of the result is equal to the width of the word being shifted. The word is shifted as if the shift register is exactly as wide as the data word. For example,

```
SLC( ACCUMULATOR<0:3>, 2 )
```

performs a 2-bit left circular shift of the 4-bit quantity in a 4-bit register.

The shift operators follow standard shift definitions. For logical shifts, each vacated bit position is replaced by zero. For circular shifts, each vacated bit is replaced by the bit shifted out (wraparound). For arithmetic shifts, vacated bits are replaced by zero (left shift) or the sign bit (right shift).

1.7 External Interfaces

A SMITE computer description may be linked to external processors or to input/output devices of the host computer through data items of storage class EXTERNAL or PORT. These external interfaces support description of a computer's connections to the outside world. In the context of emulation, external interfaces permit routines implemented in QM-1 MULTI microcode to be linked to the emulator.

A data item declared with the attribute EXTERNAL is defined to be an external processor. If the symbol is also declared with an explicit width, the external processor is thus defined to be a function rather than a subroutine. A data item declared as EXTERNAL may not be declared as an array, and may have no parameters. An external processor is invoked in the same manner that any internal SMITE processor is invoked. The differences

between external processors and the internal SMITE processors may be summarized:

1. External processors are declared by the appearance of the EXTERNAL storage class attribute in a data declaration statement. Internal processors are declared by their definition as a PROCESSOR. External processors may have no *parameters.*

2. For emulation, the microcode for an external processor is not produced by the SMITE compiler. Chapters 6 and 7 discuss the methods used to add external processors to the emulation system. SMITE internal processors are compiled to microcode as part of the computer description.

3. The external and internal processor are invoked in the same manner.

A data item declared as a PORT is an externally accessible register used as an interface cell. The PORT storage class attribute implies that a value stored by the computer into the port register is transferred as a data or control word to an external device. Similarly, a value read from a port is a status or data word transferred to the computer from the external device. Unlike other data declarations, PORT provides a direct interface with elements outside of the SMITE computer data base. PORT is *functionally identical to REGISTER* in the emulation microcode, except that an external assembly language routine is automatically activated to process the implied I/O function.

The implementation of PORT and EXTERNAL, the methods of adding external processors to the emulator, and the rules for using predefined processors provided with SMITE are described in Chapters 6 and 7.

1.8 General Rules for Coding SMITE

SMITE computer descriptions may be coded completely free field within colums 1 through 72 of the line image. Blanks may be used freely to improve readability. The following rules define the correct usage requirements for blanks in SMITE:

1. At least one blank is required to *separate alphanumeric* symbols. For example,

```
DECLAREACCUM<0:7>REGISTER;
```

is illegal because no space separates the symbol DECLARE from the symbol ACCUM. The correct coding of this statement would be

```
DECLARE ACCUM<0:7>REGISTER;
```

Note that no space is required before the symbol REGISTER, since it is not ajacent to another alphanumeric symbol.

2. Due to the lack of an underscore character in the CDC character set, the hyphen has been forced to do double duty. Because of this, incorrect interpretations of expressions involving the minus operator are possible. For example,

```
A-B;
```

is a reference to the data item or processor named 'A-B', while

```
A - B;
```

is an expression which will subtract A from B.

3. Blanks may not appear within a symbol. For example,

```
A: PROC ESSOR;
```

is illegal since a space appears in the middle of the symbol PROCESSOR. Symbols may not be split across line boundaries.

4. Statements in SMITE are invariably terminated by a semicolon. This includes all the control statements (chapter 2), expression statements, declaration statements, processor headers, and END-type statements (e.g. END IF;).

1.9 Case Study 1: The Mark 1

In this section we present a SMITE description of a complete computer, the Mark 1. The SMITE description was adapted from a description of the Mark 1 found in Bell and Newell [1]. The complete Mark 1 is as follows:

```
1.      MARK1: PROCESSOR;
2.          DECLARE
3.              M[0:8191]<0:31> MEMORY,
4.              PI<0:15> REGISTER,
```

```
5.              F<0:2> DEFINED PI<0:2>,
6.              S<0:12> DEFINED PI<3:15>,
7.           CR<0:12> REGISTER,
8.           ACC<0:31> REGISTER;
9.        DECLARE
10.          STOP EXTERNAL;
11.       DO FOREVER;
12.          BEGIN;
13.             PI <- M[CR]<0:15>;
14.             CASE F;
15.                '' 0 ''
16.                   CR <- M[S];
17.                '' 1 ''
18.                   CR <- CR + M[S];
19.                '' 2 ''
20.                   ACC <- - M[S];
21.                '' 3 ''
22.                   M[S] <- ACC;
23.                '' 4 ''
24.                   ACC <- ACC - M[S];
25.                '' 5 ''
26.                   ACC <- ACC - M[S];
27.                '' 6 ''
28.                   IF SE(ACC) < 0
29.                      THEN CR <- CR + 1;
30.                      END IF;
31.                '' 7 ''
32.                   STOP;
33.             END CASE;
34.          CR <- CR + 1;
35.       END;
36.    MARK1: END;
```

Line 1 is the processor header. It specifies that a processor is being defined, and that the name of the processor is 'MARK1'. The processor is the main processor since it is not embedded in any other processor.

Lines 2 through 8 are a declaration statement. The registers and memories of the Mark 1 will be defined in this statement.

Line 3 is the declaration of the primary memory. It is specified as an 8192-word array of 32-bit words, and is given the name 'M'.

Line 4 is the declaration of the instruction register. It is 16 bits wide, and will be referred to by the name 'PI'. Its bits are numbered sequentially from 0 to 15, starting with the leftmost bit of the register.

Line 5 is the declaration of the opcode, or function, subfield of the instruction register. The F subfield is specified to be three bits wide, and is the leftmost three bits of the PI register. The DEFINED attribute, used here to overlay F onto PI, will be explained in Chapter 3.

Lines 6 through 8 complete the register declarations. The remaining subfield of the instruction register, S, is defined, as are the program address register (CR) and the accumulator (ACC).

Lines 9 and 10 declare an external processor, STOP. In this description, no attempt was made to define the clocking or timing of instructions in the Mark 1. Rather than include lines of SMITE to show the manner in which the Mark 1 was halted, an external processor was defined to stop the computer (in some unspecified manner).

Line 11 is a SMITE control statement. Control statements will be explained in Chapter 2; the DO FOREVER statement in this example specifies that the statements in lines 12 through 35 are to be executed repetitively without end.

Line 13 is a SMITE expression statement (an expression statement is any expression terminated by semicolon, the statement termination character). The statement specifies that the upper 16 bits of the memory word at the address specified by the contents of register CR are to be transferred into register PI, thus fetching the next instruction.

Line 14 is a CASE statement, which specifies that one of the following executable statements (lines 16, 18, 20, 22, 24, 26, 28, or 32) is to be executed. The particular statement chosen is determined by the value of F.

Lines 15, 17, 19, 21, 23, 25, 27, and 31 are SMITE comments. A comment in SMITE is any string of text (except two quotes) enclosed in two single-quotes (i.e. ''). Comments may be placed anywhere that blanks may be placed.

Line 16 is the specification of the Mark 1 branch instruction, opcode 0. The contents of memory at the address defined by the contents of the S subfield of the current instruction are transferred to the program address register. Note that the program address register is incremented later in the instruction execution cycle (line 34), and so the address of the next instruction is actually M[S] + 1.

Line 18 is a relative jump, opcode 1. The sum of the program address register and the memory word at the address contained in

the S subfield of the current instruction is transferred to the program address register. As with opcode 0, the next instruction executed will be at the address one greater than the value left in CR at line 18.

Line 20 is a load complement instruction, opcode 2. The arithmetic complement of the memory word addressed by the contents of S is transferred to the accumulator.

Line 22 is a load accumulator instruction, opcode 3. The memory word at the address specified in the S subfield of the current instruction is transferred to the accumulator.

Lines 24 and 26 are the arithmetic instructions, opcodes 4 and 5. The content of the memory word addressed by S is added or subtracted from the value in the accumulator, and the result is transferred back into the accumulator.

Line 28 is the conditional skip instruction, opcode 6. If the content of the accumulator is negative, then one is added to the program address register, and that result is transferred back into the program address register. Note the use of the SE built-in function in the IF statement to specify that the accumulator contains a signed quantity. The expression in the IF statement could be written in other ways, such as

```
IF ACC<0>
   THEN ...
```

Line 32 is the halt instruction, opcode 7. The computer is halted by a call on the STOP external processor.

Line 34 increments the program address register.

Line 36 is the end of the Mark 1 processor definition.

2. Control Statements

2.1 Introduction

Control structures are provided in SMITE to form groups of statements (context blocks) and to alter the normal sequential flow of statement execution. The control structures in SMITE are designed to provide the capabilities needed to produce computer descriptions, and to adhere to good programming practices.

The need for control statements in computer description is clear even from the short case study of the Mark 1. The processing logic of a computer contains branches based on the presence/absence of a signal, the value of one or more bits, or the value of a status line. The decoding logic extracts several bits of the instruction register to determine the opcode, source register, addressing mode, or destination register. The processing logic may contain repetitive loops for shift operations, normalization, division, and multiplication. The presence of an illegal condition, or of special case logic, may cause the normal processing to be bypassed. SMITE control statements provide the capability for describing these hardware structures.

The term 'context block' refers to a group of SMITE statements forming a unit. Every SMITE control statement (except ESCAPE) forms a context block around its scope. For example, the IF statement forms a context block encompassing the THEN and ELSE statements. The DO statement forms two context blocks, one for the entire loop structure including both the loop controls and the controlled statement, and one just for the controlled statement. The CASE statement forms a variable number of context blocks, one for the entire CASE structure, and one for each substatement within the CASE statement.

SMITE also provides the capability to group multiple statements together into a context block which is treated as a single statement. This may be done with the BEGIN and END statements, e.g.

```
BEGIN;
   .
   .  (statements)
   .
   END;
```

in which case statements in the sequence are to be executed in

sequential order, first to last, or with the PARALLEL-BEGIN AND PARALLEL-END statements, e.g.

```
PARALLEL-BEGIN;
   .
   .
   .
PARALLEL-END;
```

in which case statements in the group are all to be executed simultaneously in asynchronous parallel. (Note: PARALLEL-BEGIN and PARALLEL-END are not yet compiled for asynchronous execution of the statements. The statements are executed as with BEGIN and END.)

A context block may be labeled or unlabeled. If a label is used, both the statement beginning the block and the statement ending the block must be labeled with the context block name. For example,

```
A: BEGIN;
   .
   .
   A: END;
```

or

```
B: IF ACC<0>
   THEN ...
   ELSE ...
   B: END IF;
```

or

```
C: CASE F;
   .
   .
   .
   C: END CASE;
```

2.2 The IF Statement

The IF statement selects a single statement from its component statements for execution. It specifies that the statement following the symbol THEN be executed only if a certain condition (expression) is true. If the condition is false, then either no

statement, or the statement following the symbol ELSE, is executed. The SMITE IF statement has the following two forms:

```
IF expression
   THEN statement
   END IF;
```

or

```
IF expression
   THEN statement
   ELSE statement
   END IF;
```

A context block is associated with the IF statement and may be labeled. If the label option is exercised, then both labels must be present and identical. In addition to labeling the context block these labels allow the compiler to perform verification of correct block nesting. The labeled IF statement has the forms.

```
label: IF expression
   THEN statement
   label: END IF;
```

or

```
label: IF expression
   THEN statement
   ELSE statement
   label: END IF;
```

The expression is evaluated according to the standard rules described in section 1.5, and must evaluate to a result that is one bit wide. A value of one is defined as true, and a value of zero as false.

Control passes to the statement immediately following the THEN if the expression evaluates as true (one). Control passes to the statement immediately following the ELSE if the expression evaluates as false (zero). IF no ELSE clause is present and the expression is false, then the statement immediately following the END IF statement receives control.

In either case, at most one of the statements associated with the IF statement is executed. When that statement has completed execution, control passes to the statement immediately following the END IF, unless the statement selected for execution by the IF is an ESCAPE (which causes an explicit transfer of control).

IF statements may incorporate any SMITE executable statement as the controlled statements, including BEGIN/END or another IF. For example,

```
IF expression1
   THEN statement1
   ELSE IF expression2
      THEN statement 2
      ELSE IF expression 3
          .
          .
          .
             ELSE IF expression
                THEN statement
                END IF;
             .
          .
       .
   END IF;
```

To avoid costly execution by an emulator of the above statement, it should be ordered so that the probability of expression 1 occurring is greater than the probability of expression 2 occurring, and so forth. Using this technique ensures that the least number of expression evaluations and tests is performed for each execution of the IF.

An unnecessary use of the IF statement is the following:

```
IF A = 0
   THEN ZERO <- 1;
   ELSE ZERO <- 0;
   END IF;
```

This entire IF statement can be replaced with the simpler statement:

```
ZERO <- A = 0;
```

2.3 The CASE Statement

The CASE statement provides a limited and highly disciplined branching capability. It consists of an expression, called the selector, and a list or collection of statements. The CASE statement selects one and only one statement from the collection for execution. The CASE statement is similar in usage to an

indexed jump, or to a FORTRAN computed GO TO statement, and has the following form:

```
CASE expression;
   statement
   statement
   .
   .
   .
   statement
   END CASE;
```

The CASE statement forms a context block, which may or may not be labeled. If the label option is exercised, then both the CASE and the END CASE statements must be labeled with the identical context block name. In addition to labeling the context block, labels are used by the SMITE compiler for verification of correct context block nesting. A labeled CASE statement has the form:

```
label:  CASE expression;

   statement

   statement

   .

   .

   .

   statement

   label:  END CASE;
```

The selector expression in the CASE statement is evaluated according to the standard expression evaluation rules described in section 1.5. The value of the expression is interpreted as a positive number n bits wide, where n is the width of the result. This number is used as an index into the collection of statements that follows, and the appropriate statement receives control. The first statement in the sequence is executed if the number is zero, the next one if the number is one, and so forth.

Since n bits can select one of 2**n statements, 2**n statements are required to occur between the CASE and END CASE statements. One and only one statement in the collection receives control; when that statement has finished executing, control proceeds to

the END CASE, unless the selected statement contained an ESCAPE to alter the normal flow of control.

Often situations will occur when one or more of the statements of the 2**n statement collection are to evoke no action, i.e. control is to proceed directly to the END CASE statement with no operation performed. For these instances, the NULL statement is provided. It may be coded as a statement in a CASE sequence, and will act as a do-nothing statement.

There are various alternatives to using the CASE statement. One is the nested IF:

```
IF expression1
   THEN statement1
   ELSE IF expression2
      THEN statement 2
      ELSE IF expression 3
          .
          .
          .
             ELSE IF expression
                THEN statement
                END IF;
            .
          .
        .
   END IF;
```

As discussed in section 2.2, the nested IF statement can be very inefficient in an emulation if the order that the expressions appear in the statement is not chosen carefully.

Another alternative to the CASE statement is the parallel IF construct, an example of which is:

```
PARALLEL-BEGIN;
   IF expression1 THEN statement1;
   IF expression2 THEN statement2;
   .
   .
   .
   IF expression m THEN statement m;
   PARALLEL-END;
```

In this form, all decodes are intended to occur simultaneously, and the assumption is usually made that only one of the expressions will ever be true. The order of the expressions in

this method has no impact on the efficiency of execution, since the IF statements are not nested.

When deciding on which method of decoding to use, two considerations must be made. The first is to determine whether all the conditions to be decoded are mutually exclusive. If they are not, the parallel IF is required. If the conditions are mutually exclusive, then two choices are available in addition to the parallel IF, namely the nested IF and the CASE. The CASE statement is usually preferable to the nested IF statement on grounds of clarity of the resulting description.

Instruction operation code decoding with the CASE statement is often performed in a single field within the instruction. For example:

```
DECODE:  CASE INSTRUCTION <7:6>;
    .
    .
    .
    DECODE:  END CASE;
```

The selector can also be more complex:

```
DECODE:  CASE OP-FIELD(INSTRUCTION);
    .
    .
    .
    DECODE:  END CASE;
```

where OP-FIELD is a processor defined as:

```
OP-FIELD:  PROCESSOR<3:0> (IN-VALUE);
    DECLARE IN-VALUE REGISTER;
    OP-FIELD <- ACCUM-STATE // IN-VALUE<5:4> // IN-VALUE<1>;
    OP-FIELD:  END;
```

In this case the selector is a processor that returns a value four bits wide. The value is a concatenation of two fields within the instruction and the one bit wide ACCUM-STATE.

2.4 The DO Statement

The SMITE DO statement provides the means for repetitive execution of statements or groups of statements. All forms of the DO statement are based on the concept of a controlling

statement and a controlled statement. The controlled statement may be compound (e.g. BEGIN/END) to form complex loop bodies. The general form of the DO statement is:

```
DO terminator-clauses;
   controlled statement;
```

The DO statement may be labeled:

```
label: DO terminator-clauses;
   controlled statement;
```

Two basic DO statements are defined in SMITE: DO with controlled termination clause(s), and DO FOREVER (i.e. DO with an uncontrolled terminator clause). The form with controlled terminator clause(s) provides controlled repetition (with optional iteration), while the DO FOREVER form provides uncontrolled repetition and must be used with the ESCAPE (section 2.5) statement if loop termination is desired.

2.4.1 The Uncontrolled DO Statement

The DO FOREVER statement has the following forms:

```
DO FOREVER;
   statement;
```

or

```
label:  DO FOREVER;

   statement;
```

The DO FOREVER form of the DO is equivalent to the DO WHILE 1. This form specifies an unlimited repetition of the controlled statement. Dependent upon the application, an infinite number of repetitions may be desired, such as the main loop of an emulator. In other cases, some set of circumstances may imply termination of the loop, in which case the ESCAPE statement (section 2.5) provides the method of termination.

In certain cases where a DO FOREVER has an ESCAPE, the FOREVER may be replaced by a terminator clause, and the ESCAPE eliminated:

```
LOOP: DO FOREVER;
   BEGIN;
      IF ERROR THEN ESCAPE LOOP;
      .
```

```
        .
        .
        END;
```

may be changed to

```
DO WHILE NOT ERROR;
   BEGIN;
      .
      .
      .
      END;
```

2.4.2 The Controlled DO Statement

There are three forms of controlled repetition DO statements:

1) Termination when a given condition becomes true (DO UNTIL).

2) Termination after a given condition becomes false (DO WHILE).

3) Termination when a specified iteration sequence is completed (DO FOR).

2.4.2.1 The DO WHILE Statement

The form of the DO WHILE statement is:

```
   DO WHILE expression;
      controlled-statement;
```

or

```
   label:  DO WHILE expression;
      controlled-statement;
```

The expression must evaluate to a one bit wide result. The WHILE clause executes the controlled statement as long as the specified expression is true (evaluates to a one), or until control is transferred out of the context of the loop (i.e. by an ESCAPE statement). The evaluation and test of the expression is performed once each time through the loop, and is made before the controlled statement is executed.

2.4.2.2 The DO UNTIL Statement

The form of the DO UNTIL statement is:

```
DO UNTIL expression;
   controlled-statement;
```

or

```
label:  DO UNTIL expression;
   controlled-statement;
```

The expression must evaluate to a one bit wide result. The UNTIL clause executes the controlled statement as long as the expression is false, or until control is transferred out of the context of the loop. The evaluation and test of the expression is performed once each time through the loop, and is made after execution of the controlled statement.

2.4.2.3 The DO FOR Statement

The form of the statement is:

```
DO FOR expression UP TO expression STEP expression;
```

or

```
DO FOR expression DOWN TO expression STEP expression;
```

A context block label may be used (e.g. 'label: DO').

The words UP and DOWN may be omitted, in which case UP is assumed. The phrase 'TO expression' may be omitted as well, in which case loop termination is not provided by the DO FOR statement, and if required must be separately provided using an ESCAPE statement. The STEP phrase may be omitted when the keyword UP (or no UP/DOWN keyword) is used; if omitted a value of one is used. The keyword STEP may not be omitted if the keyword DOWN is used, as a negative step is required. The phrases after 'FOR expression' may occur in any order:

```
DO FOR expression STEP expression UP TO expression;
```

or

```
DO FOR expression TO expression STEP expression DOWN;
```

or any of the other possible orderings.

The FOR clause provides capabilities similar to the FORTRAN or PL/I iterative DO loops. The UP/DOWN keyword only affects the form of the termination test, and not the incrementation of the loop counter. The action followed by a FOR clause loop is as follows:

1) The initial-value expression (the one immediately following the FOR) is evaluated. If the last evaluated operator in the expression is a data transfer ('<-'), then the evaluation is stored in the indicated data item, and all subsequent incrementations of the initial expression value will also be stored in the same data item. Note that the initial-value expression is evaluated only once, before the first execution of the controlled statement.

2) The body of the loop (the controlled statement) is executed.

3) The current value of the loop control variable is incremented by the value of the STEP expression, or by one of no step was specified. The STEP expression is only evaluated once, before the initial execution of the loop, and will not be evaluated again. If DOWN is specified, the STEP expression should evaluate to a negative number. Otherwise, loop execution occurs until the loop counter overflows and becomes negative. The loop counter is maintained in an 18-bit register by the compiler if not explicitly named, i.e.

```
DO FOR 0 TO 7;
```

causes the compiler to maintain the counter in an 18-bit register, while

```
DO FOR COUNT<0:3> <- 0 TO 7;
```

maintains and tests a 4-bit counter in the data item COUNT.

4) The resulting value (stored if so specified in the DO statement) is compared against the (optional) loop termination value. No testing is performed if no termination value is given, and the loop becomes a form of the DO FOREVER with automatic incrementation of the counter. If a termination value is given, then the test is made based on the UP or DOWN direction, and if the limit has not been exceeded then control returns to step

2. Otherwise, control passes to the succeeding statement.

2.4.2.4 The Multiple Terminator Controlled DO Statement

The tests for the DO FOR and DO UNTIL statements are made at the end of the loop (execution of the controlled statement therefore always occurs at least once), while the test for the DO WHILE statement is made at the beginning of the loop (before the controlled statement is executed). In addition to the DO WHILE, DO UNTIL, and DO FOR forms already introduced, a single controlled DO statement may contain any combination of the controlled termination phrases, as long as no more than one of each type of clause is present. For example,

```
CONTEXT-LABEL: DO FOR I <- 1 UP TO 20 STEP 1
                    WHILE J > 99;
   BEGIN;
      .
      .
      .
      END;
```

and

```
DO WHILE Q = J UNTIL R > S FOR S DOWN TO 100 STEP -5;
   BEGIN;
      .
      .
      .
      END;
```

are valid SMITE DO statements. The statement:

```
DO WHILE A >= S UNTIL PC = OP WHILE IR = ADD;
```

is not a valid SMITE DO statement because it contains two WHILE clauses. This statement could be correctly coded as

```
DO WHILE (A >= S) AND IR = ADD  UNTIL PC = OP;
```

Note the parenthesization required in the WHILE expression in order to maintain the intended sequence of evaluation.

The relative order of the FOR clause and UNTIL clause tests is dependent on the order in which they are coded. In the statement

```
DO FOR I UP TO 20 STEP 1 UNTIL Q = R;
```

the test associated with the FOR clause is performed first, and if the iteration limit has not been reached, the test associated with the UNTIL clause is performed. Again, both of these tests are performed at the end of the loop.

2.4.3 DO Statement Context Blocks

A context block is associated with the DO statement, and may be optionally labeled. This context block includes the controlled statement, no matter how complex. The end of the context block can be thought of as being just after the controlled statement, and just before the statement immediately following the range of the DO. The controlled statement will define a context block of its own, unless it is a simple statement. A few examples will explain this more clearly.

```
DO WHILE SP < PC;
   PUSH(PC);
```

The first example consists of a DO statement that has a simple statement as its controlled statement. In this case there is only one context block, which is the context block associated with the entire DO statement.

```
DO-BLOCK:  DO FOREVER;
   CONTROLLED-BLOCK:  BEGIN;
      P <- PC + 1;
      DECODE(PC);
      CONTROLLED-BLOCK:  END;
```

The above example is composed of two context blocks. The block named DO-BLOCK is the context block for the DO statement, while the block named CONTROLLED-BLOCK is the context block of the compound statement controlled by the DO statement.

```
DO FOR PC UP TO PC+5 STEP 1;
   BEGIN;
      PUSH(PC);
      DECODE(PC);
      END;
```

This example also consists of two context blocks, with the exception that this time they are unnamed. The outer block is the context block for the DO statement, while the BEGIN-END

block is the context block associated with the controlled statement.

2.5 The ESCAPE Statement

Occasionally within a computer description, situations occur in which it is desirable to immediately transfer control from the statement currently being executed to the end of some surrounding context block. The reason for the transfer may be an error condition, a "found-it" condition in a search, etc. The method of causing this transfer is the SMITE statement ESCAPE.

ESCAPE causes control to be immediately transferred to the end of a surrounding context block, i.e. a context block that lexically contains the ESCAPE statement. Escapes may not be made to a point outside the current processor. Using the ESCAPE statement with parallel context blocks and timing statements imposes additional restrictions which are described in sections 5 and 4.2, respectively.

The two forms of the ESCAPE statement are:

```
ESCAPE;
```

or

```
ESCAPE label;
```

The use of the form without the label causes control to be transferred to the end of the context block immediately surrounding the ESCAPE. For example:

```
BEGIN;
   .
   .  (statement set 1)
   .
   ESCAPE;
   .
   .  (statement set 2)
   .
   END;
```

Execution of statement set 2 is bypassed. The ESCAPE statement transfers control to the statement following the END statement.

The more common form of ESCAPE statement includes the label of

the context block to be escaped, because the ESCAPE without a label is restricted in application. For example,

```
IF DECODE-ERROR
   THEN ESCAPE;
   END IF;
```

is a no-operation statement. The context block surrounding the ESCAPE is the IF statement, and so the action of the ESCAPE is to transfer control to the statement following the IF.

The use of the form of ESCAPE with a label allows control to be transferred to the end of any surrounding context block within the current processor. This form of ESCAPE is useful for terminating the current iteration of a DO statement, for terminating loops, and for transfer of control out of nested context blocks upon the detection of an error condition.

```
BLOCK1:  BEGIN;
   .
   .
   .
   BLOCK2:  BEGIN;
      .
      .
      .
      IF OP-BAD
         THEN ESCAPE BLOCK1;
         END IF;
      .
      .
      .
      IF BAD-DATA
         THEN ESCAPE BLOCK2;
         END IF;
      .
      .
      BLOCK2:  END;
   .  (statement A)
   .
   .
   BLOCK1:  END;
.  (statement B)
.
.
```

If OP-BAD is a one, an escape is made to the end of context block BLOCK1. The next statement to be executed is that at (statement B). If BAD-DATA is a one, an escape is made to the end of

context block BLOCK2. The next statement to be executed is that at (statement A).

The interaction between ESCAPE statements and loop or DO statements allows considerable control over the action of looping in a description. Using ESCAPE and labeled context blocks, it is possible to terminate the current iteration or to terminate the loop altogether. For example:

```
BLOCK1:  DO FOR I <- 1 TO 10;
   BLOCK2:  BEGIN;
         .
         .
         .
         ESCAPE;  (1)
         .
         .
         .
         ESCAPE BLOCK1;  (2)
         .
         .
         .
         ESCAPE BLOCK2;  (3)
         .
         .
         .
         BLOCK2:  END;
```

In this example, the ESCAPE marked (1) will cause control to be transferred to the end of BLOCK2. BLOCK2 is actually the context block for the controlled statement of the DO statement, however, and so this escape terminates execution of the controlled statement for the current iteration of the loop. The DO now regains control and, depending on the then current status of the loop controls, either proceeds with another iteration of the controlled statement, or else allows control to proceed to the following statement. The ESCAPE marked (3) works in exactly the same manner, since it passes control to the end of same context block.

The ESCAPE labeled (2) works quite differently, however. It causes control to be passed to the end of the DO statement context block, which immediately causes termination of the iterative process of the loop, independent of the status of the loop controls.

```
A-BLOCK:  DO FOREVER;

   BEGIN;
```

```
.
.
.
ESCAPE;  (1)
.
.
.
ESCAPE A-BLOCK;  (2)
.
.
.
END;
```

In this example, the ESCAPE marked (1) causes control to be passed to the end of the controlled statement. The controlled statement will be executed again (from the top, of course) by virtue of the DO FOREVER. The ESCAPE marked (2) causes control to be transferred to the end of the DO statement, and thereby terminates the action of the infinite loop defined by the DO FOREVER statement.

2.6 Case Study 2: FTSC Floating Point Unit

The Raytheon Fault Tolerant Spaceborne Computer (FTSC) is a computer designed for use in future satellite applications. An emulation-oriented description developed of the FTSC computer provides an illustration of the use of SMITE for complex arithmetic operations. In this case study, we will examine the description of the floating point unit of the FTSC.

The FTSC is a 32 bit computer with the following floating point operations:

Opcode	Definition

LDNF	Complement and load operand
LDAF	Load absolute value of operand
ADDF	Floating add
SUBF	Floating subtract
MPYF	Floating multiply
DIVF	Floating divide
SRTF	Floating square root
VADDF	Vector floating add
VSUBF	Vector subtract
VMPYF	Vector multiply
VIPF	Vector inner product
VSMF	Vector scalar multiply
CFX	Floating to fixed conversion
UPF	Unpack floating point
PKF	Pack fixed point to floating
CFL	Fixed to floating conversion
JZEF	Jump if floating point 0
JNZF	Jump if not floating point 0
JPSF	Jump if positive, non-zero floating point
JMZF	Jump if negative or zero floating point

The exact operation of each floating point instruction need not be described at this time. There are a few considerations, though, which are important for understanding the FTSC description.

1) The 32 bit FTSC floating point data word is organized as a 24 bit two's complement fractional mantissa in the upper bits and an 8 bit exponent in the lower bits. The exponent is biased by X'80'. Hence, 1.0 is represented by x'40000081' (i.e. 0.5*2**1).

2) Any number which is an exact power of two is normalized to have a mantissa of X'800000' and an exponent of the power of two plus one. Numbers with only the sign bit set, such as are obtained after extracting the mantissa from floating point powers of two, are singular for certain operations, such as two's complement, and therefore precautions are sometimes required to detect these special cases.

3) Floating point 0 is represented as 00000080, a mantissa of 0 and an exponent of 0. This presents special problems for the floating point addition and subtraction operations, where the operands must be aligned.

4) The MPYF (multiply floating) instruction produces a 48 bit result. The Version 1 SMITE compiler cannot manipulate words longer than 36 bits, and therefore the description has been written to separate the result into two smaller words.

5) The floating point instructions must set an error flag if floating point exponent overflow or underflow occurs.

2.6.1 Overall Design

The first task is to identify the instructions or functions which are used as processing steps in other instructions. This division produces a natural set of floating point processors. These low level floating point instructions and functions may then be examined to identify additional common processes.

Figure 5 shows the organization of the FTSC floating point emulation in terms of SMITE processors. The top-down ordering of the floating point instructions is straightforward. In this design, the VIPF instruction could also have been implemented as a series of MPYF and ADDF instructions. The SRTF instruction has an algorithmic design very similar to the manual 'two digits at a time' method for determining decimal square roots, which precluded the use MPYF and ADDF subprocessors.

Once the basic floating point instructions were identified, common processes within these instructions were also readily apparent.

1) The results of the floating point operation are normalized. Common normalize processors perform this function.

2) The mantissa and exponent operations in the FTSC ALU require more than 24 or 8 bits of data to determine carry and overflow. The FLOATING-PREP processor sign extends the floating point data, a process patterned after the actual hardware operation. By extending the data to 32 bits, commonality with fixed point arithmetic processors is achieved.

3) The FLOAT processor converts fixed point numbers to floating point for the PKF instruction (variable exponent pack floating) and CFL instruction (fixed exponent convert to floating).

2.6.2 Top Level Processor Coding

The top level floating point processors illustrate simple DO statements and processor invocations. The vector scalar multiply (VSMF) processor was designed and coded under the following ground rules:

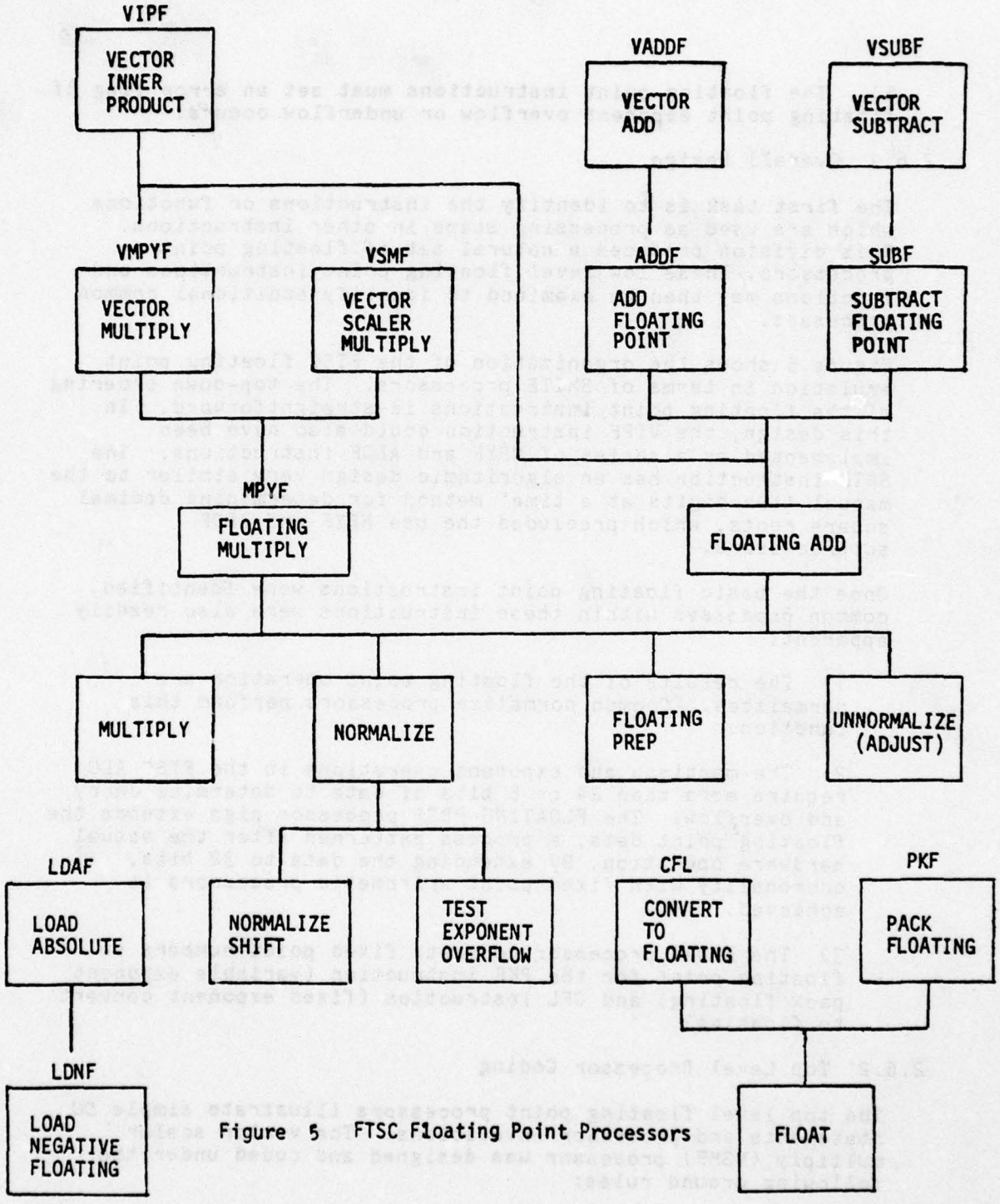


Figure 5 FTSC Floating Point Processors

1. The instruction operand is contained in the variable OPERAND, which will be destroyed by subsequent processors.

2. The processor FLOATING-MULTIPLY (no parameters) multiplies REG-OP by OPERAND, and returns the value in REG-OP.

3. The result REG-OP must be stored in GPXR[RB] after each multiply, and RB must then be incremented. RB is a 3 bit field of the instruction register. The new value of REG-OP is loaded from the GPXR array addressed by the incremented value of RB.

4. Three multiplies are performed in the scalar multiply instruction. The index RB may wrap around the address space of register array GPXR.

The FLOATING-ADD processor was designed and coded under the following constraints:

1. The two mantissas are contained in the variables OPERAND and REG-OP. The two exponents are the variables OPERAND-EXP and REG-OP-EXP.

2. The result of the floating point add operation, prior to normalization, will be stored back in REG-OP and REG-OP-EXP.

3. The operand values are moved to REG-OP and REG-OP-EXP if REG-OP equals 0. This test is necessary due to the representation of 0. The values in REG-OP and REG-OP-EXP are the correct result of the floating point add if OPERAND equals 0.

4. The OPERAND value replaces the value in REG-OP if OPERAND-EXP exceeds REG-OP-EXP by more than 23. OPERAND-EXP also replaces REG-OP-EXP in this case. This test covers the case where REG-OP adds no significance to the result.

5. The values in REG-OP and REG-OP-EXP are unchanged if REG-OP-EXP exceeds OPERAND-EXP by more than 23 (i.e. OPERAND adds no significance to the result).

6. If the exponents differ by less than 23, invoke the processor UNNORMALIZE, which aligns the mantissas REG-OP and OPERAND and forms the resulting exponent. Add the aligned mantissas.

7. Invoke the NORMALIZE processor to normalize the results.

The description written of the scalar multiply was the following:

```
1.   WORK3 <- OPERAND;                    ''SAVE OPERAND''
2.   DO FOR 1 TO 3;                       ''LOOP 3 TIMES''
3.      BEGIN;
4.         FLOATING-MULTIPLY;             ''INVOKE PROCESSOR''
5.         GPXR[RB] <- REG-OP;            ''STORE RESULT''
6.         REG-OP <- GPXR[RB<-RB+1];      ''LOAD NEXT REG''
7.         OPERAND <- WORK3;              ''RESTORE OPERAND''
8.         END;
```

Line 1 is a simple transfer operation. Line 2 is an example of a simple DO statement. The loop index is not used by the loop code, and is therefore not maintained in an explicit loop variable. Lines 5 and 6 could be recoded to use the loop index. In that case, line 2 would be recoded as

```
DO FOR I <- 0 TO 2;
```

and lines 5 and 6 would become

```
GPXR[RB+I] <- REG-OP;

REG-OP <- GPXR[RB+I+1];
```

Line 3 starts the block for the DO statement. The body of the loop exceeds on statement, and therefore is surrounded by a BEGIN/END context block. Line 8 is the END statement which closes the block.

Line 4 invokes the processor FLOATING-MULTIPLY. Line 5 moves a variable onto an array entry. (The range of address values for the array GPXR was defined in the data declaration as 0 to 7.) The contents of the variable RB supply the address.

Line 6 performs a compound operation. Following the right to left order of operations, the contents of RB plus 1 replace RB. The new value of RB is used as the address for indexing the array GPXR, and the corresponding entry is loaded into REG-OP. Instruction 7 is a simple data transfer.

The description written for the FLOATING-ADD processor was the following:

```
FLOATING-ADD:  PROCESSOR;
```

```
1.    IF (OPERAND-EXP > REG-OP-EXP + 23)
      OR (REG-OP = 0)
2.       THEN BEGIN;
3.          REG-OP <- OPERAND;
            REG-OP-EXP <- OPERAND-EXP;
4.          END;
5.       ELSE
6.          IF (OPERAND-EXP > REG-OP-EXP - 23)
            AND (OPERAND /= 0)
7.             THEN BEGIN;
8.                UNNORMALIZE;
9.                REG <- OP-REG-OP + OPERAND;
10.               END;
11.            END IF;
12.      END IF;
13.   NORMALIZE;
      FLOATING-ADD:  END;
```

This example could have been coded with 4 simple IF tests, instead of the 2 compound IF tests actually used. With either method, the branches of the IF statements should converge only at the concluding NORMALIZE operation.

2.6.3 VSMF Example

One of the examples in the previous section was the vector scalar multiply (VSMF) instruction description. In this section, the related coding is followed down to the low-level processors.

The VSMF processor invoked only the FLOATING-MULTIPLY processor. It supplied two 32-bit registers (REG-OP and OPERAND) to the FLOATING-MULTIPLY processor, and received a 32 bit result in REG-OP. The processor FLOATING-MULTIPLY was constructed under the following design constraints:

1. The variables REG-OP and OPERAND are operated upon, and the result is stored back in REG-OP.

2. The MULTIPLY processor used for the fixed point multiply instruction is used to multiply the mantissas. MULTIPLY performs a 32-bit multiplication of the variables REG-OP and OPERAND and returns the result in REG-OP and EX. The actual multiplication algorithm uses the absolute values of the variables.

3. The result is normalized and stored in REG-OP and EX.

4. A zero result is stored as 00000080.

5. Exponent overflow and underflow must be tested.

The description of FLOATING-MULTIPLY is:

```
FLOATING-MULTIPLY:  PROCESSOR;
   FLOATING-PREP;
   REG-OP <- SLL(REG-OP, 8);
   MULTIPLY;
   IF REG-OP = 0
      THEN REG-OP <- X'80';
      ELSE BEGIN;
         REG-OP-EXP <- REG-OP-EXP + OPERAND-EXP;
         DBL-NORMALIZE;
         END;
      END IF;
   FLOATING-MULTIPLY:  END;
```

As a consequence of requirements 2 and 5, the first operation, performed by an invocation of FLOATING-PREP, is to separate the mantissa and exponent of each input into two variables. The mantissas are sign extended, and the characteristic is padded with zeroes.

The call to FLOATING-PREP is the first code in the FLOATING-MULTIPLY routine. The value of REG-OP is now tested for zero after a left shift of 8 bits, and a floating point zero value is stored if required. Given unnormalized multipliers, however, it is conceivable that the upper 24 bits of the product are zero and the lower bits non-zero. This process, although intuitively inaccurate, is the one described because it is a modeling of the actual process of the computer.

In the case of a non-zero product, the mantissa has been calculated and is stored in the appropriate variables. The pre-normalization exponent is calculated. The 48 bit result must then be normalized. Although this appears to be a cumbersome operation, only the more significant data word is operated upon. If it is all zeroes or ones, the less significant data word is moved up and the normalization count is incremented by 24. The normalization count of the upper word is found. The corresponding number of bits is shifted out of the lower word. The exponent of the result is adjusted by the normalization count and tested for overflow/underflow. The mantissa and exponent are then merged back into the result variable.

The important concept in this case study is that complex arithmetic processes may be described with SMITE. Analysis of

the operations performed in the actual hardware will usually lead the SMITE programmer to a natural modularization and flow of processors which closely models the operations performed in the hardware.

3. Storage Sub-Structure and Operation Widths

3.1 Introduction

Many classes and types of memory devices and structures exist in digital computer systems. Memories vary widely with respect to access time, storage capability, read and/or write capabilities, interfaces, error detection, etc. SMITE provides a means of declaring data items, the attributes in the data item declaration statement, that describes the differing structural aspects of different memory devices. The active characteristics of some memory devices, such as error detection and correction or associative access, are properly described with SMITE processors. This chapter presents the full storage attribute facility provided in the SMITE language.

In addition, SMITE provides a basic abstract data format capability to define data patterns which are independent of their physical location. For example, an instruction format may successfully be defined as subfields of the instruction register. Floating point data, however, is more properly defined using the abstract data format technique, since floating point data may occur in many different locations of the machine, and is not bound to a specific register. The abstract data format definition attribute is also presented in this chapter.

3.2 Data Item Attributes

As described in section 1.4, a data item declaration within a DECLARE statement consists of the data item name followed by its attributes. The complete form is as follows:

```
identifier array width class defined
```

where 'array' represents an array attribute ('[...]'), 'width' represents a width attribute ('<...>'), 'class' represents a storage class attribute (e.g. REGISTER or MEMORY), and 'defined' represents a defined attribute (section 3.3).

Some (or all, when the DEFAULT facility of section 3.4 is used) of the attributes may be omitted. Those attributes which do appear must appear in the order shown above. The defined and array attributes are always optional, while the width and class attributes are optional only if provided by a DEFAULT. The width

and array attributes are described in sections 1.4.4 and 1.4.5 respectively. The defined attribute is described in section 3.3; the remainder of this section is devoted to the description of the storage attribute.

The various types of storage attributes defined in SMITE are used to describe the physical characteristics of the storage elements (register, memory, I/O port, etc.) and to define abstract data formats. The following storages attributes are provided in SMITE:

REGISTER - indicates a storage element likely to be used with high frequency. It is otherwise equivalent to MEMORY, and therefore corresponds to the intuitive notion that a register is a fast memory.

MEMORY - indicates an undistinguished storage element in the sense that the use of the data item does not constrain the element to have any special properties (e.g. I/O port, speed, etc.). Mass storage devices could (potentially) be represented using huge arrays having the MEMORY attribute; however, SMITE does not support any virtual memory or secondary storage capability to implement this feature if the limits of the QM-1 are exceeded.

SWITCH - corresponds intuitively to switches on the operators console of the target computer. It represents a read-only binary storage device which is to be connected to the operator interface mechanism.

LIGHT - is the display analog of SWITCH. It is a write-only binary storage device which is to be output through the operator interface mechanism.

PORT - denotes a register which is an I/O interface. PORT functions identically to REGISTER in the resultant microcode, except that an external microcode routine is automatically activated to process the implied I/O function.

FLAG - denotes a 1 bit wide REGISTER.

EXTERNAL - indicates that the named data item is an external microcode closed procedure. If a width definition is attached to the name, the procedure is assumed to be a function; no array attribute is allowed with a storage element that has an EXTERNAL attribute.

CLOCK - defines the storage element name to be used to reference the internal clock (section 4). The clock is kept

in a SMITE internal location which imposes the following restrictions:

1. only one clock declaration is permitted,

2. the declaration must not have an array attribute,

3. the clock may not be DEFINED (section 3.3) onto another storage element, and

4. the clock must have a width attribute 36 bits wide.

One other class attribute, DATA, is available to define abstract data formats which are referenced in several different physical locations. One typical use of DATA would be to define the floating point number format. No storage is allocated for DATA items; reference to these items is always qualified by a simultaneous reference to a register or other storage elements.

The interpretation of a complete qualified reference (of the form STORAGE.DATA) is as follows. For a parent DATA item (i.e. one which is not DEFINED onto another item), if the bit width of the storage and DATA items correspond, then they are directly overlapped. If the bit widths differ, however, then if the DATA item bit definitions are numbered in the same direction as the storage item, and if the subfield indicated by the DATA item bit definition exists in the storage element, then the corresponding overlay is made. For DATA fields which are DEFINED onto other DATA items, the parent item is first overlayed onto the storage item as explained above. The DEFINED subfield as positioned by the parent is then used. For example, given the declarations

```
DECLARE
   REG<15:0> REGISTER,
   FULL-WORD<0:15> DATA,
      LOW-BYTE<0:7> DATA DEFINED FULL-WORD<8:15>,
      HIGH-BYTE<0:7> DATA DEFINED FULL-WORD<0:7>,
   MIDDLE-BYTE<11:4> DATA;
```

then

```
REG.FULL-WORD
```

is equivalent to the entire register,

```
REG.LOW-BYTE
```

selects the rightmost 8 bits of the register,

```
REG.HIGH-BYTE
```

selects the leftmost 8 bits of the register, and

```
REG.MIDDLE-BYTE
```

selects the middle 8 bits of the register.

3.3 The DEFINED Attribute: Storage Overlays

SMITE allows a tree structured data definition technique, whereby subfields may be defined as overlays on a parent data item. Attributes of the parent data item (except an array attribute) are passed to the subfield unless overridden by explicit declarations for the subfield. The overlay of one data item on another is achieved through the use of the DEFINED attribute in the data declaration statement for the subfield. A DEFINED attribute has the following form:

```
DEFINED parent-subfield-reference
```

In the following example,

```
DECLARE
   ACCUM<7:0> REGISTER,
   SIGN FLAG DEFINED ACCUM<7>;
```

the data item ACCUM is the parent, and SIGN is the subfield.

The parent subfield reference is a description of the portion of the parent to be overlayed by the subfield. It consists of the parent data item name, an array attribute, and a width attribute. If either or both of the attributes are omitted, the entire scope of the parent is assumed.

The two data items must be compatible, i.e. the total the number of bits referenced in the parent must be equal to the total number of bits in the subfield, an element of the subfield must not cross a word boundary of the parent, and if multiple words of the parent are referenced then the entire width of each parent word must be referenced. The following overlay operations are permitted:

1. A word of the parent may be broken up into one or more subfields, or into an array of subfields. For example, the following declarations are permitted:

```
DECLARE
   DATA-REGISTER<0:31> REGISTER,
      ADDRESS-FIELD<0:23> DEFINED DATA-REGISTER<8:31>,
      BYTES[0:3]<0:7> DEFINED DATA-REGISTER;
```

2. A parent array may be broken up into an array of smaller sized words. For example,

```
DECLARE
   PARENT-ARRAY[0:7]<0:31> REGISTER,
      SUBFIELD-ARRAY[0:15]<0:15> DEFINED PARENT-ARRAY;
```

3. A portion of a parent array's address space may be overlayed by a subfield. For example,

```
DECLARE
   PARENT-ARRAY[0:0'77777']<0:47> MEMORY,
      GENERAL-REGISTERS[0:7]<0:47> DEFINED
      PARENT-ARRAY[0:7];
```

Rather complicated structures may be constructed, if required, with the DEFINED attribute. For example,

```
DECLARE
   DWM[0:X'FFFF']<0:63> MEMORY,
      WM[0:X'1FFFF']<0:31> DEFINED DWM,
         HWM[0:X'3FFFF']<0:15> DEFINED WM,
            BM[0:X'7FFFF']<0:7> DEFINED HWM,
               BTM[0:X'3FFFFF']<0> DEFINED BM,
      INTERRUPT-VECTOR[0:7]<0:31> DEFINED WM[X'300':X'307'];
```

The larger-to-smaller requirement for the structure hierarchy must be strictly observed. For example,

```
DECLARE
   MAIN[0:1023]<0:15> MEMORY,
      SUBFIELD<0:31> DEFINED MAIN[0:1];
```

is illegal since the item SUBFIELD is required to cross a word boundary. One way to correctly define this structure is as follows:

```
DECLARE DOUBLEWORD[0:511]<0:31> MEMORY,
   MAIN[0:1023]<0:15> DEFINED DOUBLEWORD,
      SUBFIELD DEFINED DOUBLEWORD[0];
```

Note the use of omitted attributes in the DEFINED attribute. The array and width subfield reference attributes are missing from the parent description for the MAIN subfield, and therefore are

obtained from the data declaration of the parent item. The attributes obtained are [0:511] and <0:31> respectively. The width subfield reference attribute is also omitted from the SUBFIELD declaration, and so the full width of the parent is referenced.

The above example also illustrates the inheritance of attributes from the parent to the subfield itself. In the declarations of both MAIN and SUBFIELD, the storage class attribute is omitted, and therefore MEMORY is inherited from the parent DOUBLEWORD. In the declaration of SUBFIELD, the width attribute is omitted, and so a width of <0:31> is inherited from its parent, DOUBLEWORD. Note that the length attribute, '[0:511]' is not inherited by SUBFIELD from DOUBLEWORD.

Class attributes declared in subfield data items must be identical with the class attribute of the parent. For purposes of comparison, REGISTER and MEMORY are considered equivalent.

An interaction occurs between the block structuring produced by nested processors and global data references, and the DEFINED attribute. In the following example:

```
A: PROCESSOR;
   DECLARE Q <15:0> REGISTER;
   J: PROCESSOR;
      DECLARE B FLAG DEFINED Q <15>;
      DECLARE Q <7:0>;
      .
      .
      .
      J: END;
   .
   .
   .
   A: END;
```

The item B is a subfield of the global data item Q instead of the data item Q local to processor J, because the declaration of the local data item Q has not been seen at the time B is declared.

```
A: PROCESSOR;
   DECLARE Q<15:0> REGISTER;
   J: PROCESSOR;
      DECLARE Q<7:0>;
      DECLARE B FLAG DEFINED Q<15>;
      .
      .
      .
```

```
   J: END;
   .
   .
   .
A: END;
```

The only difference between this and the last example is the order of the DECLARE statements in processor J. The definition of the subfield B is now in error, since it refers to the local Q as the parent and no bit numbered 15 exists in the Q local to J. To avoid confusion, a local parent should always be declared immediately prior to its subfields.

3.4 Defaults

Frequently, a number of data items will be declared in a SMITE computer description with identical attributes. For example:

```
DECLARE
   A<7:0> REGISTER,
   B<7:0> REGISTER,
   C<7:0> REGISTER,
   D<7:0> REGISTER,
   E<7:0> REGISTER,
   F<7:0> REGISTER;
```

SMITE allows the declaration of common attributes by the declaration of the special data item DEFAULT. Any time a data declaration is made and an attribute is missing, the DEFAULT declaration is searched for the missing attribute. The use of defaults simplifies the computer description, and permits concentration on the unusual characteristics of a data item declaration.

A declaration of defaults is made by declaring the special identifier DEFAULT, using the same data declaration statement form used for all other data item definitions. The DEFAULT declaration may appear any place that a data declaration is valid. The name DEFAULT is a SMITE reserved word, however, and may never be used as an actual data item, nor is it ever allocated space by the compiler. The following example

```
DECLARE
   DEFAULT <31:0> REGISTER;
```

defines the default width attribute to be a field 32 bits wide

numbered from 0 to 31 and from right to left. It also defines the default storage class to be REGISTER.

There exist no pre-set defaults in SMITE, and therefore in the absence of a default, the following example

```
DECLARE
   ACCUM <7:0>,
   XTENSION REGISTER;
```

is in error because both of the data declarations are incomplete. However, this example

```
DECLARE DEFAULT <7:0> REGISTER,
   ACCUM,
   XTENSION <0:3>;
```

is acceptable. The data item ACCUM is declared to be type REGISTER, eight bits wide, and has bit numbering 0 to 7 from right to left. The data item XTENSION is declared to be type REGISTER, four bits wide, and has bit numbering 0 to 3 from left to right.

Only one DEFAULT declaration may exist in a SMITE description. That DEFAULT should be in the main processor, preferably as the first declaration.

3.5 Case Study 3: Intel 8080 Storage Declarations

The storage declaration section of the Intel 8080 computer description provides several illustrative examples of data item definition. The 8080 storage declaration is the following:

```
1.  DECLARE DEFAULT<7:0> REGISTER,
2.     PROD-FLAGS<35:0>,
3.        STEP-FLAG FLAG DEFINED PROD-FLAGS <35>,
4.        INTERRUPT-IR<7:0> DEFINED PROD-FLAGS<25:18>,
5.        INTE FLAG DEFINED PROD-FLAGS<17>,
6.        INTERRUPT FLAG DEFINED PROD-FLAGS<16>,
7.        PC<15:0> DEFINED PROD-FLAG <15:0>,
8.     REGS[0:3]<15:0>,
9.        REGISTERS[0:7]<7:0> DEFINED REGS,
10.           B DEFINED REGISTERS[0],
11.           C DEFINED REGISTERS[1],
12.           D DEFINED REGISTERS[2],
13.           E DEFINED REGISTERS[3],
```

```
14.            H DEFINED REGISTERS[4],
15.            L DEFINED REGISTERS[5],
16.            STATUS DEFINED REGISTERS[6],
17.               CARRY FLAG DEFINED STATUS<0>,
18.               PARITY FLAG DEFINED STATUS<2>,
19.               AUX-CARRY FLAG DEFINED STATUS<4>,
20.               ZERO FLAG DEFINED STATUS<6>,
21.               SIGN FLAG DEFINED STATUS<7>,
22.            A DEFINED REGISTERS[7],
23.               CARRY-A<8:0> DEFINED REGS[3]<8:0>,
24.         SP<15:0>,
25.         B-PAIR<15:0> DEFINED REGS[0],
26.         D-PAIR<15:0> DEFINED REGS[1],
27.         H-PAIR<15:0> DEFINED REGS[2],
28.         IR,
29.              IR-OPERAND-SELECT<2:0> DEFINED IR<2:0>,
30.              IR-DEST-SELECT<2:0> DEFINED IR<5:3>,
31.              IR-SUB-FUNCTION<2:0> DEFINED IR<2:0>,
32.              IR-TEST-SELECT<2:0> DEFINED IR<5:3>,
33.         OP-REG,
34.         HOLD-A,
35.         OP-PAIR<5:0>,
36.         TEST-FLAG,
37.         HOLD-REG,
38.         MEM[0:4095] MEMORY;
39.      DECLARE OP-STEP EXTERNAL,
40.              OP-HALT EXTERNAL,
41.              OP-ERROR EXTERNAL,
42.              OP-NOTSIM EXTERNAL,
43.              IN-PORT PORT,
44.              OUT-PORT PORT;
```

In line 1, the default width and data attributes are declared. As a result of this statement, when no width is declared for a variable, the default width <7:0> is used. When no data attribute is used, the default attribute of REGISTER is used.

Lines 2 through 7 define the variable PROD-FLAGS and the substructures of that word. PROD-FLAGS is defined as a 36 bit word, the sign bit being represented as bit 35 and the least significant bit as bit 0. The statement

```
DECLARE
   PROD-FLAGS<0:35>;
```

would also have defined a 36-bit word, but with a different bit representation (right to left in increasing order). Three variables overlayed on PROD-FLAGS are defined with the FLAG

attribute, each data item thereby being specified as being one bit wide. Line 3 could be recoded equivalently as

```
DECLARE
   STEP-FLAG<0> DEFINED PROD-FLAGS<35>;
```

The result of these definitions is to form 5 subfields on the parent word. When any of these variables is read, the specified bits are extracted from the parent word for use in the expression. When the variables are stored into, only the specified bits in the parent word are altered. When the parent word is used in an expression, the entire word is used regardless of the substructure defined on the word.

Line 8 defines a register array of 4 words length and 16 bits width. The first element of the array is designated as REGS[0], and the last element as REGS[3].

Line 9 defines a substructure of the array REGS. REGISTERS is an array of 8 words length and 8 bits width. REGISTERS is a valid substructure of REGS since it does not cross word boundaries.

The registers could have been defined as separate independent variables. However, the instruction decoding algorithm often uses a 3 bit field to select the register. Grouping the registers in an 8 word array permits the description to access the proper emulated register by an access to the register file REGISTERS. Similarly, the registers are allocated within 16 bit register pairs because that structure allows the computer description to adhere to the architecture more closely.

Lines 10-23 define alternate names for the registers to improve the clarity of the computer description. For example, the register C is a renaming of REGISTER[1]. The C register can also be referred to as a subfield of REGS[0] or B-PAIR.

Lines 17-21 define the substructure of the STATUS register. The variable CARRY, for example, is equivalent to STATUS<0>, REGISTERS[6]<0>, and REGS[3]<8>. Referring to the status bits by the individual name improves the readability of the description.

Lines 22 and 23 provide an example of an extended variable. REGS[3] is allocated in one parent word. REGISTERS [6] and REGISTERS [7] form a substructure of the parent word, and so the registers STATUS and A occupy contiguous subfields in the parent word. The bit immediately adjacent to the sign bit of A is CARRY. Arithmetic operations which reference the subfield A//CARRY may instead be written to reference CARRY-A, which holds

both CARRY and A. In addition to offering a certain compactness of expression, this technique improves the speed of emulation when referencing the concatenated field.

Lines 25-27 provide an alternative representation of REGS[0], REGS[1], and REGS[2] for improved readability.

Lines 28-32 define IR, the instruction register, and its substructure. Notice that IR-OPERAND-SELECT and IR-SUB-FUNCTION define the same bit positions. The use of two names reflects the different usages of those bits during the instruction decode. For certain opcodes, those bits select the destination register; in other cases, they provide a secondary opcode.

The Intel 8080 computer description does not use the DATA attribute to define substructure. The definitions in lines 28-32 could be recoded as follows:

```
IR,
INSTRUCTION<7:0> DATA,
   OPERAND-SELECT<2:0> DATA DEFINED INSTRUCTION<2:0>,
   DEST-SELECT<2:0> DATA DEFINED INSTRUCTION<5:3>,
   SUB-FUNCTION<2:0> DATA DEFINED INSTRUCTION<2:0>,
   TEST-SELECT<2:0> DATA DEFINED INSTRUCTION<5:3>;
```

With this form of instruction format specification, for example, all uses of the variable IR-OPERAND-SELECT in the computer description would be replaced by IR.OPERAND-SELECT. The advantage to this technique is clearer, for example, if a data structure called SIGN-BIT is defined as INSTRUCTION<7>. Then the sign bit of an 8 bit variable, say register A, may be referenced by A.SIGN-BIT rather than the less descriptive A<7>.

Line 38 defines the primary memory array.

Lines 39-44 define the external processors and I/O port registers.

4. Timing Specification and Control

4.1 Introduction

For many applications, it is sufficient to describe the data transfer and control operations of a computer. There is an inherent value in the computer description, and an emulator compiled from such a description can be used to debug and execute a large class of software.

In other applications, it is necessary to consider timing relationships in the computer description. Each instruction requires a certain amount of time to execute on the target computer. Interrupts may occur based upon the real-time clock. Input operations require a given amount of time to complete, and status lines may change within certain intervals after the initiation or completion of input. Output operations may require a fixed minimum interval between successive commands.

When a complete data processing system is to be simulated, the need for an emulator clock is even more apparent. For many command and control applications, the computer is used to control specialized hardware. An accurate simulation of the hardware requires knowledge of the exact time that the computer issues various commands.

A timing-free description of the instruction operation of a computer specifies the architecture of a general family of computers. For a particular computer in the architecture family, there are many possible timing options, such as the instruction clock frequency, memory cycle time, I/O device access times, I/O transfer rates, and clock interrupt frequency.

One application of an emulator is the prediction of the effect on system performance of a particular timing change. For example, an emulation could be used to answer questions such as "What is the effect of doubling the instruction clock frequency for a computer when a particular program is run?" The execution time for a particular instruction may depend on both the instruction clock frequency, the memory cycle time, and I/O access delays and transfer rates. The new execution time of each instruction can be calculated. The degree of change in instruction execution times probably vary for different instructions, depending on the operations performed within each instruction. Any given program has a specific instruction mix, and will not necessarily benefit proportionately from the increase in CPU speed. The use of an emulation provides the means to experiment to determine the

effect of the proposed change. Similar tradeoffs may be performed for memory cycle time or various I/O options.

4.2 The IN Statement

The SMITE construct used to express timing relationships is the IN statement. IN defines the amount of time required to perform the next SMITE statement. The syntax of IN is the following:

```
IN expression statement
```

The statement so prefixed with an IN statement is specified to require 'expression' time units to execute. The time increment, of course, may be a constant or some variable expression. The units of time are left to the discretion of the programmer. The units may be unspecified ('clock counts'), or else the expression may be followed with one of the following noise words:

Unit	Abbreviation
SECONDS	S
MILLISECONDS	MS
MICROSECONDS	US
NANOSECONDS	NS

These time units are noise words. No form of units checking, conversion, or other significance is attached to the word by the compiler. It is the programmers responsibility to ensure that all IN statements in a description utilize the same units.

The current clock value, as updated by execution of timed statements, may be referenced through a 36-bit data item declared with the CLOCK attribute. The clock is initialized to zero when emulation execution is initiated.

The IN statement forms a context block surrounding the timed statement, and may be labeled:

```
label: IN expression statement
```

No escape may be made past the IN context block; an escape to the label on the IN statement will result in the clock being updated as specified by the expression.

The clock is updated at the completion of the statement. If one or more IN statements are nested within another IN statement, the

inner statements define the subintervals in the interval defined by the outer statement. The expression should not evaluate to a negative result. The clock will remain unaltered should such an attempt be made.

Should nested IN statements advance the clock past the time specified for completion of an outer statement (through incorrect description of timing), then this later time will remain in the clock after execution of the outer statement completes. This procedure is employed to prevent the occurrence of negative time intervals.

4.3 The Amount of Timing Detail in a Computer Description

Several options exist as to the level of timing detail present in a SMITE computer description. For example, an IN statement may surround the invocation of an entire processor which executes the current instruction, in which case the timing expression would be an average execution time, or, at the other extreme, individual timings may be specified for every lowest level operation in the description.

In general, lower level timing specification requires more information about the timings of the computer due to the increased level of detail present. Very low level timing specifications tend to be constants, while as the level of timing specification increases more complicated expressions tend to occur in the attempt to accurately model the aggregate effect of conditionals or loops nested under the IN statement. There is no 'correct' level of detail for inserting timing into a computer description. The level of detail employed depends on the intended use of the description and the amount of detailed timing information available.

4.4 Case Study 4: A Shift Unit

For a very large number of computers, the execution time of a shift instruction depends upon the number of bits shifted. In this case study, we consider three cases of description of a shifter unit. In all cases, the shift performed will be left logical, shifting the data in a register A by a shift count in a register N.

1. The execution time is 4 units plus 1 for every bit shifted (4 + N).

2. The execution time is 12 units plus 3 for every bit shifted (12 + 3N).

3. The execution time is 4 units when N=0, 5 units for N=1 or 2, and 6 units when N=3 or 4 (two bit at a time shifts).

For the first case, there are two possible alternatives:

```
IN 4+N A <- SLL(A,N);
```

or

```
IN 4 SHIFT-COUNT <- N;
DO WHILE SHIFT-COUNT /= 0;
   IN 1 PARALLEL-BEGIN;
      A <- SLL(A,1);
      SHIFT-COUNT <- SHIFT-COUNT - 1;
      PARALLEL-END;
```

In the second case, the two methods become

```
IN 12+N+N+N A <- SLL(A,N);
```

and

```
IN 12 SHIFT-COUNT <- N;
DO WHILE SHIFT-COUNT /= 0;
   IN 3 PARALLEL-BEGIN;
      A <- SLL(A,1);
      SHIFT-COUNT <- SHIFT-COUNT - 1;
      PARALLEL-END;
```

In the third case, the two methods become

```
IN 4 + SRL(N+1,1) A <- SLL(A,N);
```

and

```
IN 4 SHIFT-COUNT <- N;
DO WHILE SHIFT-COUNT > 1;
   IN 1 PARALLEL-BEGIN;
      A <- SLL(A,2);
      SHIFT-COUNT <- SHIFT-COUNT - 2;
      PARALLEL-END;
IF SHIFT-COUNT /= 0
   THEN IN 1 A <- SLL(A,1);
```

```
END IF;
```

5. Parallelism

5.1 Introduction

Within a computer there are a number of operations which occur in parallel or overlap each other. Oftentimes by making simplifying assumptions about the operation or relative independence of processes, it is satisfactory to describe a computer in terms of serial steps. In the case of advanced processors or computer systems, it is more natural and efficient to describe the system in terms of parallel processes. SMITE provides the parallel context block to support the description of parallel processes.

5.2 The PARALLEL-BEGIN and PARALLEL-END Statements

The PARALLEL-BEGIN and PARALLEL-END statements define a context of parallel processes in the same way that BEGIN and END define a serial context. Each statement in a parallel context is executes simultaneously and asynchronously with all the other statements in the parallel context. The parallel processes start at the same time; the parallel context completes execution when the last process finishes.

The syntax of the PARALLEL-BEGIN and PARALLEL-END statements is similar to BEGIN and END. A label may be used with the parallel statements. The same label must be used for both PARALLEL-BEGIN and PARALLEL-END, if a label is used. The full set of executable SMITE statements is allowed within the parallel context.

(Note: The SMITE compiler does not currently generate microcode to execute the statements of a parallel context simultaneously. The context is compiled as if it were serial, i.e. as if the surrounding statements had been BEGIN and END.)

5.3 Parallel Timing and Control Flow

The interval required to execute the statements in a parallel context is the maximum execution time for any one statement, not the sum of the processor execution times. Within a parallel context, an IN statement causes a temporary clock to be incremented. At the conclusion of the parallel context, the

maximum temporary clock value updates the aggregate clock. This reflects the fact that the parallel processes which complete first are suspended until the last process is finished.

An ESCAPE statement within a serial context causes subsequent statements to be bypassed. In a parallel context, the ESCAPE statement can only bypass statements within the parallel process in which it lies. The remaining parallel processes are not affected by execution of the ESCAPE. It is not possible to escape from a parallel context out of that context. This constraint coincides with the intuitive understanding that the parallel processes operate independently of each other and an ESCAPE should be a local condition.

5.4 Case Study 5: Instruction Pre-Fetch

A simple example of parallelism is the instruction prefetch logic of a computer. The emulation of each target computer instruction can be regarded as four consecutive steps: The instruction fetch, the instruction decode, the instruction operand fetch, and the instruction execution. In many computers, the instruction fetch overlaps the other steps to produce faster execution times. The technique of reading the next instruction while executing the current one is referred to as instruction prefetching.

One common prefetch convention, sometimes called instruction pipelining, is to load the instruction register with the next instruction, and to begin the fetch of the following instruction location at the end of the current instruction. Ignoring complicating factors, such as multi-word instructions or branch instructions, the computer may be described in terms of its basic steps as follows:

```
COMPUTER-WITH-PREFETCH:  PROCESSOR;
   INSTRUCTION-FETCH;
   DO FOREVER;
   PARALLEL-BEGIN;
      BEGIN;
         DECODE;
         OPERAND-FETCH;
         INSTRUCTION-EXECUTE;
         END;
      INSTRUCTION-FETCH;
      PARALLEL-END;
   COMPUTER-WITH-PREFETCH:  END;
```

The first complicating factor is multi-word instructions. At the time of the instruction fetch, reading the next word in memory is begun on the assumption that this will be the next instruction. For certain computers, this memory word (current instruction location + 1) is part of the instruction or the instruction operand.

The computer description may then be recoded in the following manner:

```
COMPUTER-WITH-PREFETCH:  PROCESSOR;
   INSTRUCTION-FETCH;
   DO FOREVER;
      BEGIN;
         DECODE;    ''MAY SET NEED-NEXT-WORD''
         IF (NEED-NEXT-WORD)
            THEN PARALLEL-BEGIN;
               NEED-NEXT-WORD <- 0;
               DECODE2;   ''DECODE NEXT WORD''
               INSTRUCTION-FETCH;
               PARALLEL-END;
            END IF;
         IF (OPERAND-MEMORY-NEEDED)
            THEN PARALLEL-BEGIN;
               OPERAND-MEMORY-NEEDED <-0;
               OPERAND-FETCH;
               INSTRUCTION-FETCH;
               PARALLEL-END;
            ELSE OPERAND-FETCH2;
            END IF;
         PARALLEL-BEGIN;
            INSTRUCTION-EXECUTE;
            INSTRUCTION-FETCH;
            PARALLEL-END;
         END;
   COMPUTER-WITH-PREFETCH:  END;
```

The third case occurs when the current instruction is a branch instruction which forces the program out of sequence or when an interrupt occurs. The instruction available in the pipeline is then not the next instruction to execute. For the computer description to reflect this capability, we may recode the first lines of code as follows:

```
COMPUTER-WITH-PREFETCH:  PROCESSOR;
   INSTRUCTION-FETCH;
   DO FOREVER;
      BEGIN;
         IF BRANCH OR INTERRUPT
```

```
    THEN INSTRUCTION-FETCH;
    END IF;
DECODE;
    .
    .
    .
```

6. Input/Output and Operator Interface

6.1 Introduction

For many intended applications, the SMITE programmer is not concerned with input/output operations of the emulated computer. From the standpoint of a CPU description, the I/O channels or ports are very similar to memory. The CPU provides data to a port, and sometimes also provides an enabling signal to the device. A black-box operation transmits the data to the particular output device. Status registers or ports may be read by the CPU, but the changes in the status lines occur independently of the CPU. Similarly, from the CPU viewpoint, input data appears on a channel from an external source which itself determines the values and the time of arrival.

There are a number of significant advantages in limiting the I/O description to the CPU-to-port interface. First, it is a realistic description of the CPU operation. The CPU relinquishes control at the I/O devices through these ports. Second, the computer description is then independent of the I/O devices present in the computer system. Finally, SMITE has no capability of linking directly with actual I/O devices. Since interface software is required between the SMITE code and an I/O device driver it is not unreasonable for the interface software to simulate the operation of the external device.

For an emulation of the CPU and some I/O devices, software linking the SMITE microcode to device drivers or the operating system is required. This linkage is also important for operator interfaces and the invocation of microcoded routines. This section will discuss the interface of the SMITE object code to other software executing on the QM-1.

6.2 PROD/TASK Interface to SMITE

PROD and TASK are the two basic elements of the QM-1 operating system. TASK provides the overall system control, processes interrupts, schedules tasks, handles inter-task linkages, and communicates with I/O devices. For most applications, including SMITE, PROD is the primary task running under TASK. PROD allows operator communication with an emulator, provides display and control commands, and provides the emulator's interface with the outside world.

A number of PROD capabilities are dependent on the organization of emulated memory, the desired data formatting, and the particular emulator communication request. Rather than build these emulator-dependent features directly into PROD, a small emulator-dependent program, referred to as the Program Interface Driver (PID), interfaces the general PROD capabilities to the specific emulator.

In the implementation of SMITE, PROD and TASK are identical for every emulation. The PID and emulator represent unique elements of the system. SMITE emulators interface to PROD and TASK through SASS, the SMITE Application Support Software. The PID is incorporated into the version of SASS constructed for a particular emulation.

It is important to understand the method of communication between TASK, PROD, and the emulator. Figure 6 shows the relative organization of main store and control store in the system. Each of the three tasks, TASK, PROD, and the SMITE emulation, can access its own main memory and the main memory of lower priority tasks. Each task addresses the first word of its main store as word 0 (i.e. relative main store addressing) and therefore must be cognizant of the relative offset of the lower tasks. Each task addresses or transfers to the control store address of a lower priority task through a fixed addressing scheme which both tasks observe. Lower priority tasks usually communicate with higher priority task through SYSTEM instructions.

Of particular importance to the SMITE emulator is the Recall SYSTEM instruction, which transfers control back to PROD for further action. The SMITE program requests an external process to occur by a particular SYSTEM Recall instruction. Ultimately a PROD overlay is invoked to process the SMITE call. This routine is a PROD-level task, using the PROD register set and main store addressing. This method of communication frees the SMITE compiler from providing explicit linkages with PROD and from protecting the emulator registers from inadvertent changes in PROD. The SYSTEM Recall instructions generated by the SMITE compiler provide the linkage to PROD and TASK.

6.3 External Functions

A processor declared as an external processor in the SMITE computer description is implemented as a PROD overlay. A recall SYSTEM instruction is generated by the SMITE compiler to link to

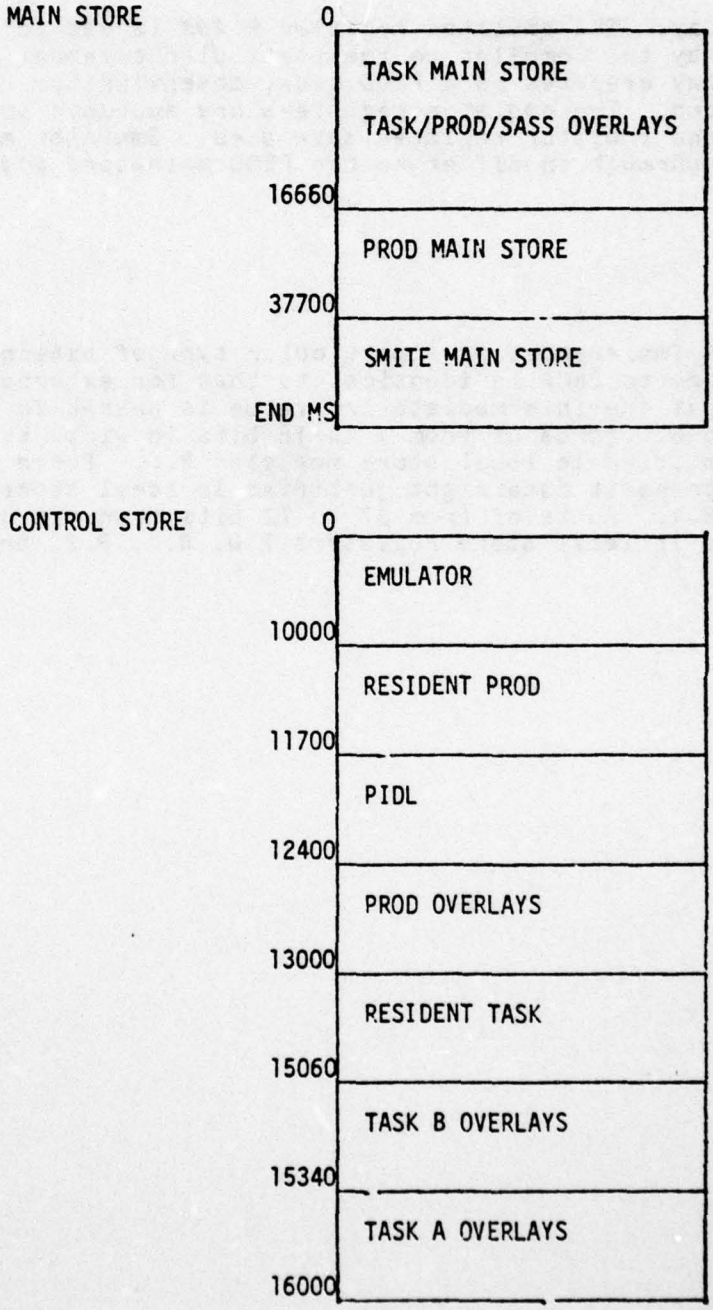


FIGURE 6 QM-1 MEMORY ALLOCATION

the overlay. The emulator register R.ADR is set to the ordinal assigned by the compiler to the particular external function. The overlay executes as a PROD task, observing the PROD register conventions. The emulator registers are accessed indirectly through the emulator register save area. Emulator main store is accessed through an offset to the PROD mainstore addressing.

6.4 Ports

A port is implemented as a particular type of external function. The linkage to PROD is identical to that for external functions, except that the intermediate I/O value is passed in the emulator local store. Ports of from 1 to 18 bits in width transmit data right justified in local store register R.0. Ports of from 19 to 36 bits transmit data right justified in local store registers R.0 and R.1. Ports of from 37 to 72 bits transmit data right justified in local store registers R.0, R.1, R.2, and R.3.

7. SASS: SMITE Application Support Software

7.1 Introduction

Once the SMITE computer description compiles successfully, it is ready to test on the QM-1. The process of transporting the object code of the compiler and executing on the QM-1 is relatively straightforward. A special system, the SMITE Application Support Software (SASS), is used to load and test emulators developed with SMITE. SASS is an augmented version of the TASK/PROD operating system which provides:

1. Additional commands to load and test SMITE emulators,

2. Interface for PROD control and display commands,

3. A predefined set of external functions, and

4. An 8080 emulator state display.

The SMITE programmer's interface with SASS does not begin the first day he tests the emulator on the QM-1. By considering the SASS capabilities during the computer description, the programmer can significantly improve his test efficiency. This section will focus on four particular aspects of SASS:

1. The requirements imposed on the SMITE computer description by the SASS implementation,

2. The preparation of magnetic tapes to load the emulator and SMITE main memory,

3. The steps used to load, run, and debug SMITE emulators, and

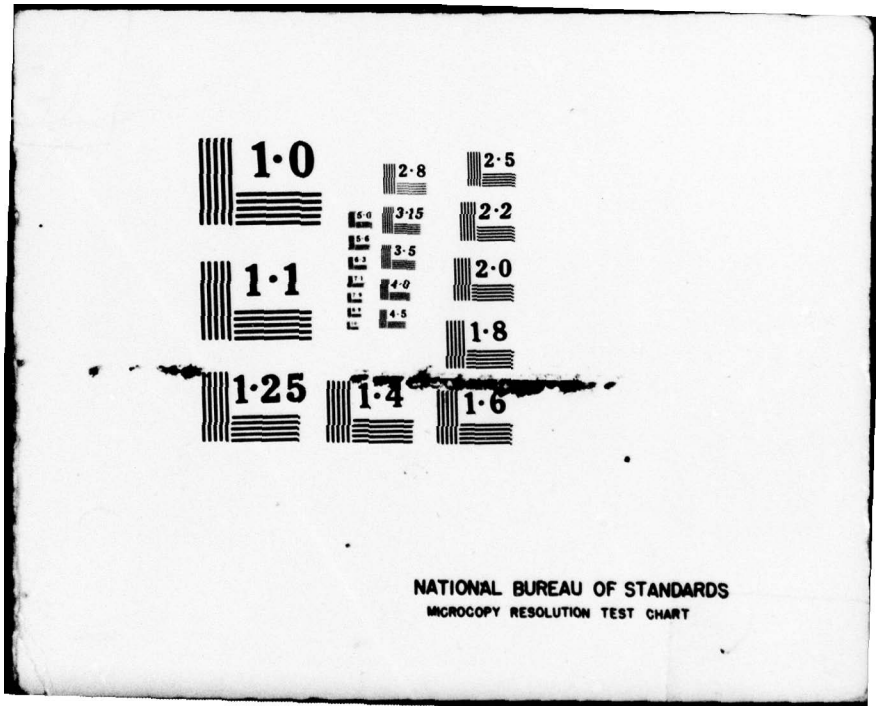
4. The procedures necessary to modify SASS for particular emulators.

7.2 SMITE Computer Description Development Considerations

When writing the computer description, the SMITE programmer should be aware of conventions established by the implementation of SASS. SASS requires knowledge of the location of the step flag, emulated program counter, an I/O control register, emulated

AD-A049 038 TRW DEFENSE AND SPACE SYSTEMS GROUP REDONDO BEACH CALIF F/G 9/2
SMITE REFERENCE MANUAL.(U)
NOV 77 F30602-77-C-0089
UNCLASSIFIED TRW-30417-6002-RU-00 RADC-TR-77-364 NL

2 OF 3
AD
A049038


1·0
1·1
1·25
2·8
3·15
3·5
4·0
4·5
1·4
2·5
2·2
2·0
1·8
1·6
NATIONAL BUREAU OF STANDARDS
MICROCOPY RESOLUTION TEST CHART

memory in the SMITE data base, and the order of the external function declarations in the computer description. These items are predefined in Version 1 SASS, thus establishing interface conventions for the SMITE programmer to follow.

7.2.1 Step Flag

The first SMITE data item should be declared as a 36 bit wide data item, such as

```
DECLARE
   PROD-FLAGS<35:0>;
```

Bit 35 of this variable is the emulator step flag. The PROD STEP command causes the SMITE step flag to be set. For the step to then occur, the computer description must call the SASS OP-STEP external function (which, in turn, resets the step flag). For example:

```
DECLARE
   PROD-FLAG<35:0>,
      STEP-FLAG FLAG DEFINED PROD-FLAG<35>,
      OP-STEP EXTERNAL;
DO FOREVER;
   BEGIN;
      IF STEP-FLAG
         THEN OP-STEP;
         END IF;
      .
      .  ''PROCESS EMULATED INSTRUCTION''
      .
      END;
```

The emulator will cycle through emulated instructions. Each execution of the PROD STEP command will cause the execution of one target machine instruction. After the emulator has been checked out, this STEP implementation may be used to debug software executing on the emulator.

Although the name STEP and the description in PROD imply that the emulator cycles through the emulation of one target computer instruction, SMITE STEPs may occur several times for an emulated instruction, or may be spaced several emulated target machine instructions apart. For example:

```
DECLARE
   PROD-FLAG<35:0>,
      STEP-FLAG FLAG DEFINED PROD-FLAG<35>,
      OP-STEP EXTERNAL;
```

```
DO FOREVER;
   BEGIN;
      .
      .
      .
      IF STEP-FLAG
         THEN OP-STEP;
         END IF;
      .
      .
      .
      IF STEP-FLAG
         THEN OP-STEP;
         END IF;
```

This code may be used to step the emulation through one or more SMITE statements. The primary advantage of this technique is that the emulator may be debugged at the SMITE description level with little examination of the MULTI code produced. The drawback is the large amount of code which must be inserted and then removed from the computer description.

7.2.2 Emulated Program Counter

The emulated program counter must be declared as the rightmost bits of the first SMITE variable. For example, to declare a 16-bit program counter,

```
DECLARE PROD-FLAGS<35:0> REGISTER,
   PC<15:0> REGISTER DEFINED PROD-FLAGS<15:0>;
```

The program counter is used for the SMITE state display, the PROD BREAK and TON commands, and the SASS SPC command. (Note: The SMITE programmer may change the location of the emulated program counter if the corresponding variable in the SASS Program Interface Driver List (PIDL) is also modified. Reference [12] defines the structure and operation of the PIDL.)

7.2.3 Emulated Memory

Most SMITE computer descriptions will contain a data item representing the primary memory of the computer. For convenient inspection and modification of the primary memory, PROD includes the M (memory inspection) and SM (memory modification) commands. An emulation which has been fully integrated into SASS, such as the Intel 8080, includes a modified PIDL (and therefore the M and SM commands) to properly reference primary memory. SASS has been supplied

with two complete PIDLs: one for the Intel 8080, and one for general use.

The SMITE programmer may activate either the Intel 8080 PIDL (ACT/8080) or the general purpose PIDL (ACT/SMITE). The general purpose version makes no assumptions about the location of emulated memory in the SMITE data base; the PROD memory display and memory modify commands accept octal addresses and data, and refer to QM-1 addresses in the SMITE main memory data base. The user must translate from emulated addresses to QM-1 addresses.

When the Intel 8080 version of SASS is activated, however, the memory display and modify commands commands refer to the Intel 8080 primary memory. Addresses and data are input in hexadecimal, and the display only outputs the 8-bit Intel 8080 data image.

7.2.4 I/O Interface Register

SASS provides a limited interface between the QM-1 keyboard/CRT and the SMITE computer description. This interface was developed to serve console interface needs of the Intel 8080 emulation, but is available for use by other emulations if required.

Two external functions are defined for console I/O: IN-PORT, and OUT-PORT. Their function is defined in section 7.2.6. In order to interface properly with these two functions, the emulator must provide an I/O interface control register. The emulator must store control information in the interface register prior to invoking either external I/O function. The location of the I/O interface register is (decimal) words 12 and 13 in the SMITE data base (the variable IR in the Intel 8080 computer description).

7.2.5 Order and Operation of Predefined SASS External Functions

A number of useful external functions are defined for SMITE computer descriptions by SASS. The linkage between SMITE and SASS is maintained by ordinal. The first external function or port declared in the SMITE description is assigned the ordinal 0 by the compiler, the next one 1, and so forth. This ordinal is passed to SASS by the SMITE emulator when the function is to be called. The predefined functions and their ordinals are as follows. The specific names shown are irrelevant, and are only used for mnemonic purposes.

OP-STEP

Ordinal 0. OP-STEP clears the emulator step flag, places the emulator in step mode, and returns control to PROD. Control is returned to the emulator as the completion of the external call after any of a number of PROD commands, including STEP and GO.

OP-HALT

Ordinal 1. OP-HALT places the emulator in halted mode, as after a target computer halt instruction, and returns control to PROD. Control is returned to the emulator as the completion of the external call after any of a number of PROD commands, including STEP and GO.

OP-ERROR

Ordinal 2. OP-ERROR displays an 'emulator error' message, and returns control to PROD. Control is returned to the emulator as the completion of the external call after any of a number of PROD commands, including STEP and GO.

OP-NOTSIM

Ordinal 3. OP-NOTSIM displays an 'illegal instruction' message, and returns control to PROD. Control is returned to the emulator as the completion of the external call after any of a number of PROD commands, including STEP and GO.

IN-PORT

Read emulated keyboard status or input character.

OUT-PORT

Read emulated CRT status or output character.

7.2.6 SASS I/O Support

SASS at this time has a limited I/O support package. The I/O control register (section 7.2.4) is set prior to invoking the I/O port. The lower bit of the control register and the ports are used in the following manner.

Out-Port Bit = 0

Outputs ones complement (7 bits) of value passed on the CRT. For example,

```
DECLARE
   OUT-PORT<0:7> PORT;
```

and

```
   OUT-PORT <- ACCUM<7:0>;
```

In-Port Bit = 0

Input ones complement (7 bits) of the value input on the user keyboard. For example,

```
DECLARE
   IN-PORT<0:7> PORT;
```

and

```
   ACCUM<7:0> <- IN-PORT;
```

In-Port Bit = 1

A status value is returned. Referencing the declaration

```
DECLARE
   IN-PORT<0:7> PORT;
```

Bit 5 will be reset at all times to indicate that the CRT is available for character output; bit 7 will be set if and only if a character is available for input.

7.3 Format of SMITE Load Tapes

The object code generated by the SMITE compiler, as well as any software which runs on the emulator, is transported to the QM-1 on magnetic tape. The procedures used to generate these magnetic tapes are determined by the host QM-1 installation. This section specifies the format of these transfer tapes.

7.3.1 SMITE Emulator Load Tape

The SMITE load tape consists of a series of 256 18-bit word records written on either a 7 track or 9 track tape. The first record contains 3 words of control information and 253 unused words. Word 0 is the load address of the emulator (currently zero). Word 1 is the length of the control store image (currently the last control store location loaded).

Word 2 is the initial execution address of the emulator. The remaining records contain the image of the emulator control store starting from the load address.

The compiler object file generated on the CDC-6000 series computers is in the exact logical format required for the load tape with one important exception. Each word of the compiler object file is 60 bits wide, and contains information in the right most 18 bits. The tape generation program must extract the 18 bits of information and write a packed binary tape consisting of 4608-bit (256 x 18) records. For 9-track tapes, each physical record therefore contains 576 frames.

7.3.2 Target Software Load Tape

At some point, the emulator is ready to execute target computer software. The PROD command LOADMT is used to load the output of the target computer assembler, compiler, or loader into QM-1 main store for execution by the emulator; the target software is loaded from the Target Software Load Tape.

The format of this tape is an absolute QM-1 core image of the SMITE emulator main store data base, including the PROD interface word, the machine registers, the primary memory, and any other declared SMITE variables. The full 18 bits of data must be loaded for each QM-1 word, regardless of the SMITE definition.

The tape generator program is responsible for mapping data onto main store images. For instance, if each emulated word is 8 bits wide, and is stored in the right most 8 bits of a QM-1 36-bit memory word pair, such as is the case for the Intel 8080 emulation, the tape generator program must supply 28 leading zero bits for each 8-bit data byte so that the tape image is 36 bits of data with emulator data in the lower 8 bits.

The tape is arranged in 512 word physical records (512 x 18 = 9216 bits) and written in odd parity. A 9-track tape physical record, therefore, contains 1152 frames of data.

7.4 SASS Commands

7.4.1 PROD Commands

SASS provides the full capability of the PROD commands, along with several SMITE-related features. The following table lists each available command and its function.

Command	Function
ACT	Activate PIDL
BREAK/	Target code breakpoint
C/,C+,C-	Display control store
CLEAR	Reinitialize
CLR/	Preset Main Store
CLRMBP/	Clear one microbreakpoint
CLRMBP	Clear all microbreakpoints
CONS/	Console to emulator
DLS	Display local store
DMP	Print main store
DSMITE	Write control store to tape
DSP	Display overlay
DUMPMT	Dump main store to tape
GO	Start emulator
HALT	Halt emulator
LIB/	Library
LS/	Set local store
LOADMT/	Load main store
LOCK	Lock Commands
LSMITE/	Load SMITE Emulator
M/,+,-	Display Main Store
MBP/	Set micro-breakpoint
MOD/	Modify overlay
N	Trap at next microinstruction
PRINT	Print CRT display
PRO	Proceed from microbreakpoint
PROD	Return console to PROD
Q/	Display PROD main store
SC/	Change control store
SM/	Change main store
SQ/	Change PROD memory
STEP	Emulator step
TON/	Trace next instructions
UNLK	Unlock control store and local store commands
@	Repeat last command
Carriage Return	Step
Space	Generate emulator state display

The PROD commands are described in the Nanodata PROD User's

Guide [12]. The SMITE related inputs are defined in the following paragraphs.

7.4.2 ACT

The PROD command ACT/SMITE or ACT/8080 is used to activate the SMITE emulator capabilities and to enable the SMITE related commands. The choice of whether ACT/SMITE or ACT/8080 is used depends on whether the general purpose PIDL or the Intel 8080-specific PIDL, respectively, is desired.

7.4.3 LSMITE

The command LSMITE/UNIT/FILE loads an emulator tape into control store. The emulator tape, described in section 7.3.1, contains the initial load address, final load address, and initial execution address. The file is specified as an octal number beginning at 0 for the first file on the tape, the same convention that the LOADMT command utilizes. If no file is specified, the emulator is loaded from file 0 of the tape. For example,

```
LSMITE/1/2
```

will cause control store to be loaded with the emulator stored as the third physical file on tape unit 1.

7.4.4 DSMITE

The command DSMITE/UNIT/FILE/end-address writes the contents of control store onto a tape in the format of a SMITE load tape. The initial load address, initial execution address, and the nominal end address are saved from the preceding LSMITE command. The end-address input may be used to override the nominal end address in case a patch area is created necessitating that additional control store be dumped. The file number is an octal number starting at 0. Breakpoint instructions are restored to the original code prior to the tape dump; if breakpoints are to be retained as active in the running emulation, then the user must reset them.

An example of DSMITE is:

```
DSMITE/0/0/10000
```

Control store will be dumped to tape on the first file of tape unit 0 in a format compatible with LSMITE. The addresses dumped will be 0 to 10000 (octal). If no end address had been

required, then any of the following equivalent forms could have been used:

```
DSMITE/

DSMITE/0

DSMITE/0/

DSMITE/0/0
```

7.4.5 N

The N (next) command may be used to step microinstruction execution within the SMITE emulator (but emphatically not within SASS or PROD!!). The emulation must be stopped at a microbreakpoint for the N command to be accepted. The N command determines the address of the next microinstruction to be executed, inserts a microbreakpoint, and initiates emulator execution. The microbreakpoint will be executed immediately following execution of the current microinstruction. Use of the N command has the advantage over the microinstruction step facility of the QM-1 hardware that all of the facilities of PROD are available between microinstructions. As with normal PROD microbreakpoints, N command microbreakpoints may be cleared and the emulator restored to a free-running state by the PRO command.

7.4.6 CLRMBP

The command CLRMBP may be used to clear microbreakpoints. The command CLRMBP/address clears a specified microbreakpoint; the command CLRMBP clears all microbreakpoints. The SMITE state display contains a list of all microbreakpoints. For example,

```
CLRMBP/12345
```

clears a microbreakpoint at address 12345 (octal), whether it is the currently active one or not. If microbreakpoints are set at addresses 100,1047, and 3600, then all three of them would be cleared by the command

```
CLRMBP
```

CLRMBP/ is rejected if there is no microbreakpoint set at the specified address.

7.4.7 SPC

The command SPC/address sets the program counter of the emulated computer to the specified value. For the Intel 8080 SASS system, the address is input in hexadecimal. For the generalized SASS system, the address must be input in octal.

7.5 SASS Modification

SASS is a generalized support system for SMITE emulators. Although the general purpose version of SASS is quite useful for the initial testing of an emulator, additions to or modifications of SASS are often desirable at later stages for:

1. Specialized input/output support,

2. Emulation-unique state display,

3. Special commands to alter emulation variables (registers, switches, controls) directly rather than through SM command, and

4. New external functions.

This section will discuss the SMITE-unique requirements for any system modification. The programmer should refer to the Nanodata documentation for PROD protocols [12], the Multi instruction set [10], and NCS operation [13].

7.5.1 SASS Files

The SASS files delivered should not be modified by the SMITE programmer in order to maintain configuration control over the system. The proper method for modifying SASS is to create equivalent routines which perform the modified function. Table 7.5.1 describes the files used to create SASS.

7.5.2 RADC:GEN

RADC:GEN is an NCS EXEC file which controls the generation of the SASS system cartridge tape. New user supplied routines should be added to this file as additional inputs to the NCS program PREP, which is executed by the RADC:GEN file. The corresponding file for the QM-1 installation at TRW is SASS:GEN.

TABLE 7.5.1 SASS Program Files

SASS:GEN	PREP file for TRW SASS
RADC:SASS	PREP file for RADC SASS
SASS:ASM	Assembly file for SASS
S:DSM, PD:DSM	Source, binary for DSMITE command
S:LSM, PD:LSM	Source, binary for LSMITE command
S:SMI, PD:SMI	Source, binary general SMITE PIDL
S:SMC, PD:SMC	Source, binary SMITE command driver
S:INIREG, PD:INIREG	Source, binary for emulator initialization
S:SPC, PD:SPC	Source, binary for SPC command
S:SMR, PD:SMR	Source, binary for SMITE user recall
S:CBK, PD:CBK	Source, binary for CLRMBP command
S:URO, PD:URO	Source, binary for SMITE externals
S:SDB, M:SDB	Source, binary for SMITE display buffer
S:SMD, PD:SMD	Source, binary for SMITE display
S:NXT, PD:NXT	Source, binary for Next command
S:SMDEFS, SMDEFS	Source, binary for SMITE Definition File
N:SMITES	Source SMITE Nanocode
N:SMITE	Object SMITE Nanocode
N:SMITEDF	Binary SMITE Nanocode

7.5.3 PID

The first 37 words of the Program Interface Driver are allocated by PROD and SASS for specific functions. The initial word is the entry point instruction for calls by the ACT function. The next 32 words are the standard PROD user's PIDL (Program Interface Driver List). The next 4 words are reserved for SASS storage. The user routine to process an emulator step must remain consistent with the SMITE conventions.

7.5.4 State Display

The SMITE display routine should serve as a model for other user display drivers. The code contained in file S:SMD is the SMITE state display driver, and is table driven to a large extent. Modifying the tables in S:SMD and the corresponding data in the main store preset code, S:SDB, is a simple approach to creating new state displays. For state displays requiring drastic revisions to the state display, the programmer should consult the PROD reference manual [12].

7.5.5 Command Driver

The routine S:SMC is the source file for the SMITE command driver. New commands are added to the command table along with the overlay name of the particular command processor. New files should be created built upon S:SMC to add emulation-unique commands.

7.5.6 User Recalls (SMITE externals and ports)

The routine S:SMR contains the processing for user recalls. The routine S:UR0 contains the processing for the SMITE external functions (user recall 0). The emulator local store register R.ADR contains the index, starting at 0, to the particular SMITE external function or port. The processing for the step function must be consistent with the SMITE code and the PID step routine.

References

[1] Bell, C. G., and A. Newell, Computer Structures: Readings and Examples, McGraw-Hill, Inc., 1971.

[2] Boehm, B. W., D. W. Kosy, and N. R. Nielson, "Simulation Aids for Designing Integrated Information Systems: The ECSS Language," Astronautics and Aeronautics, November 1972, pp 68-74.

[3] Knudsen, M. H., PMSL, An Interactive Language For System-Level Description and Analysis of Computer Structures, Carnegie-Mellon University (NTIS AD 762 513), 1973.

[4] Hill, F. J., "Introducing AHPL," Computer, December 1972 (Vol. 7, No. 12), pp 28-30.

[5] Chu, Y., "Introducing CDL," Computer, December 1972 (Vol. 7, No. 12), pp 31-33.

[6] Siewiorek, D. "Introducing ISP," Computer, December 1977 (Vol. 7, No. 12), pp 39-41.

[7] Press, B. and R. K. McClean, "The Flexible Analysis, Simulation, and Test Facility: A Practical Software First Capability," IEEE NAECON '76 Record, pp 264-268.

[8] McClean, R. K., and B. Press, "The Flexible Analysis, Simulation, and Test Facility: Diagnostic Emulation," TRW Software Series TRW-SS-75-03, October, 1975.

[9] Hilbing, Col. F. J., "The RADC Computer Architecture Program,"

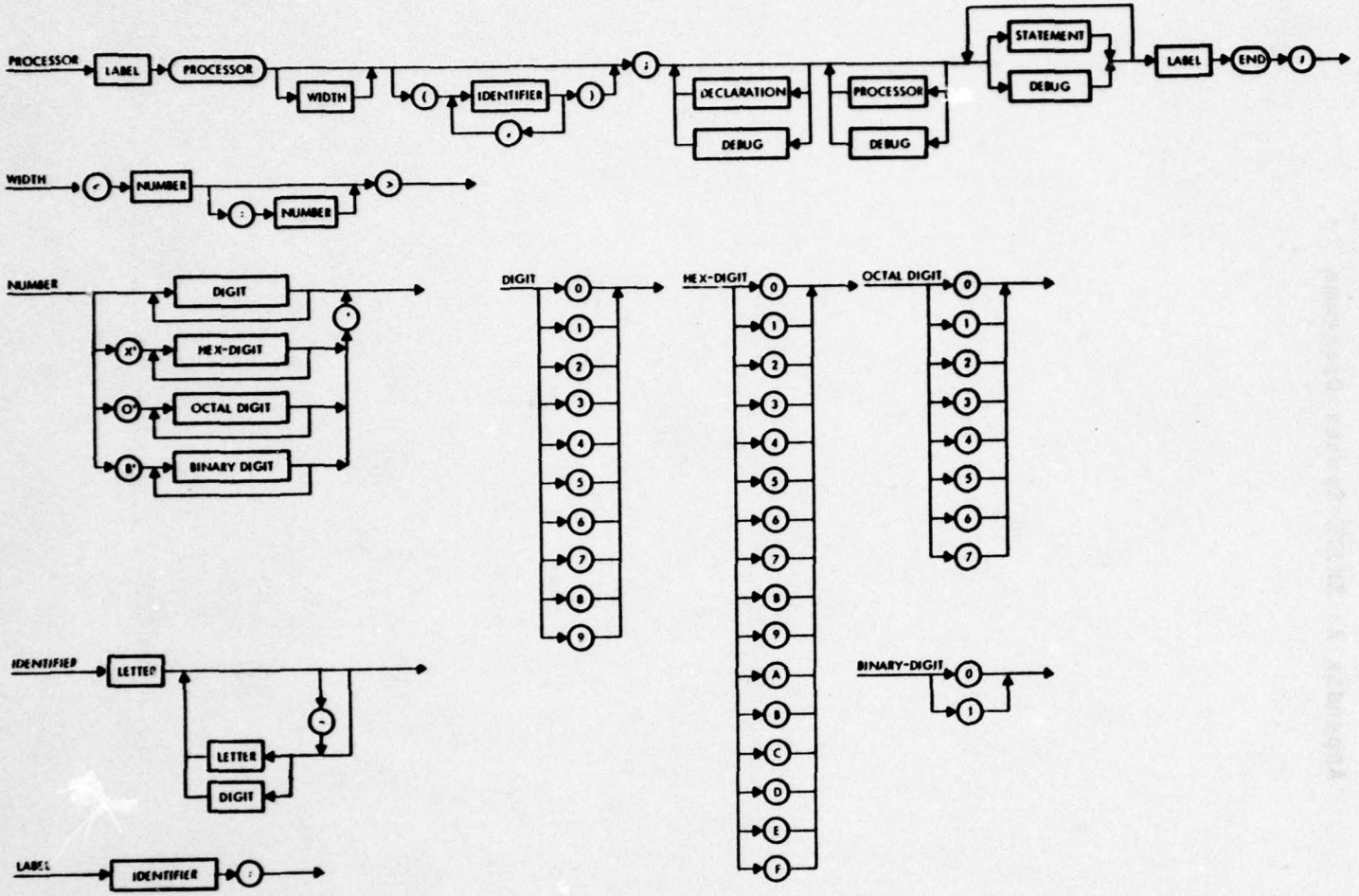
[10] MULTI Micromachine Description, Nanodata Corporation, Williamsville, New York.

[11] QM-1 Hardware Level User's Manual, Nanodata Corporation, Williamsville, New York.

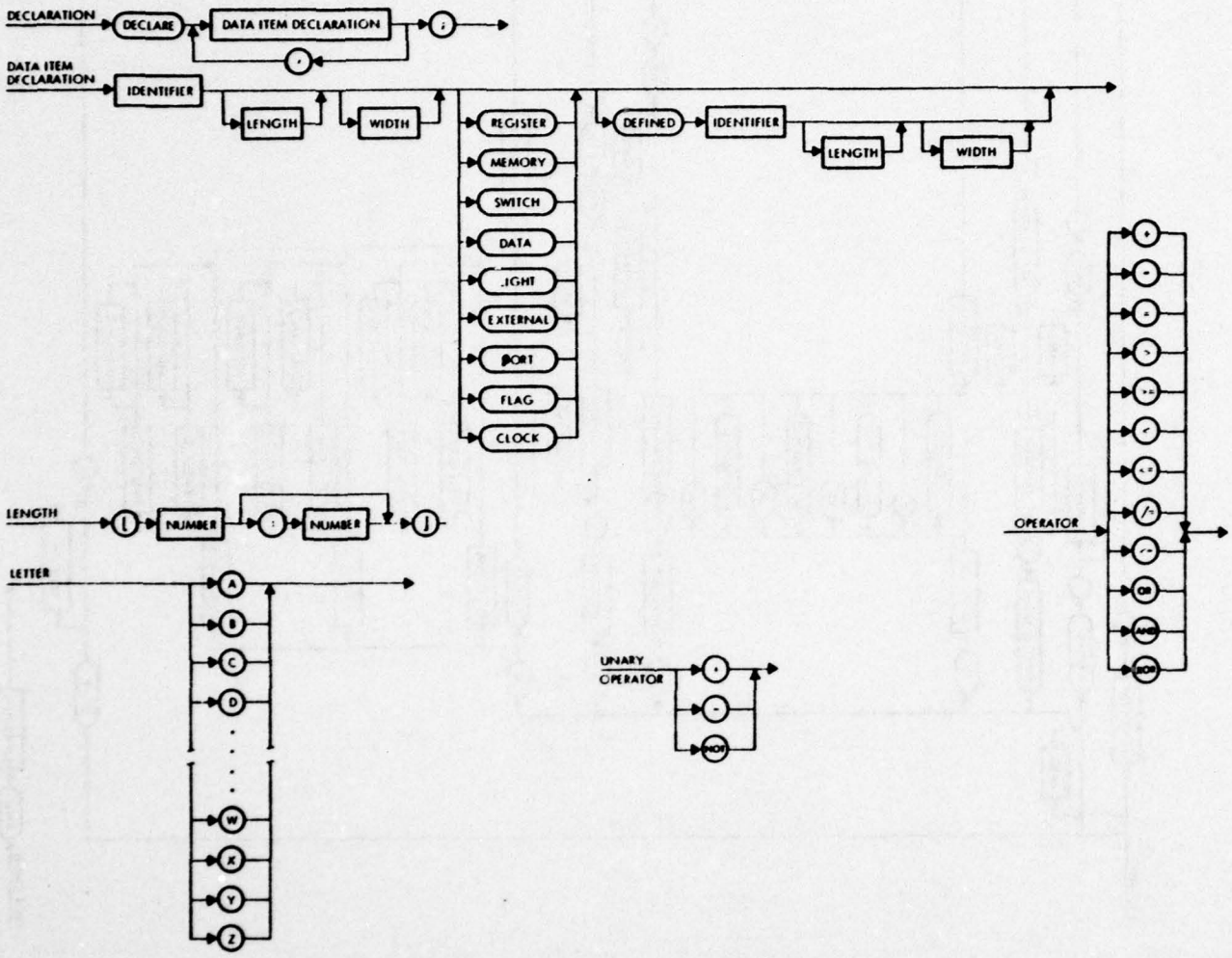
[12] Programmable Run-time Operators Display (PROD) Manual, Nanodata Corporation, Williamsville, New York.

[13] NCS User's Manual, Nanodata Corporation, Williamsville, New York.

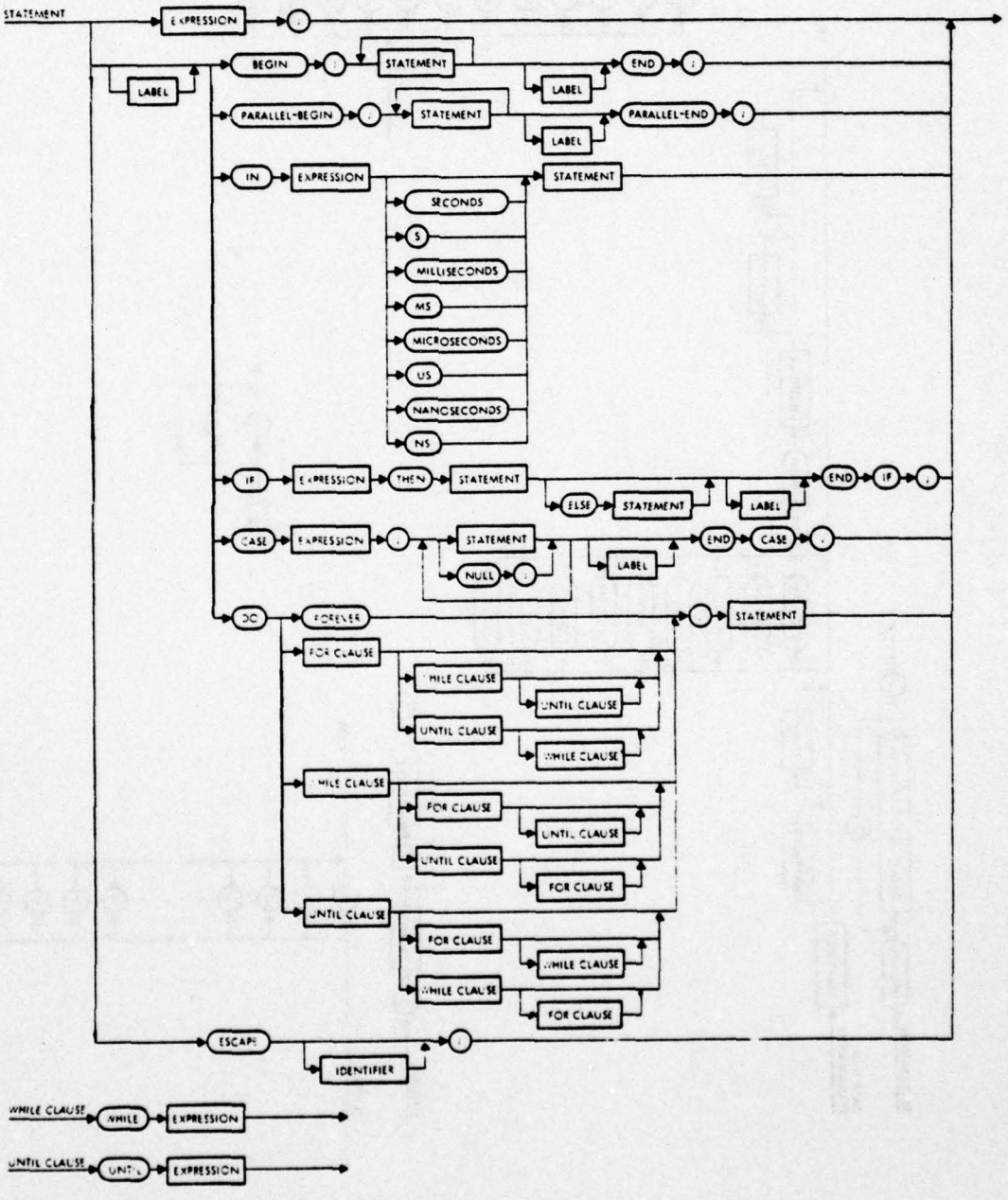
Appendix A: SMITE Syntax Diagrams

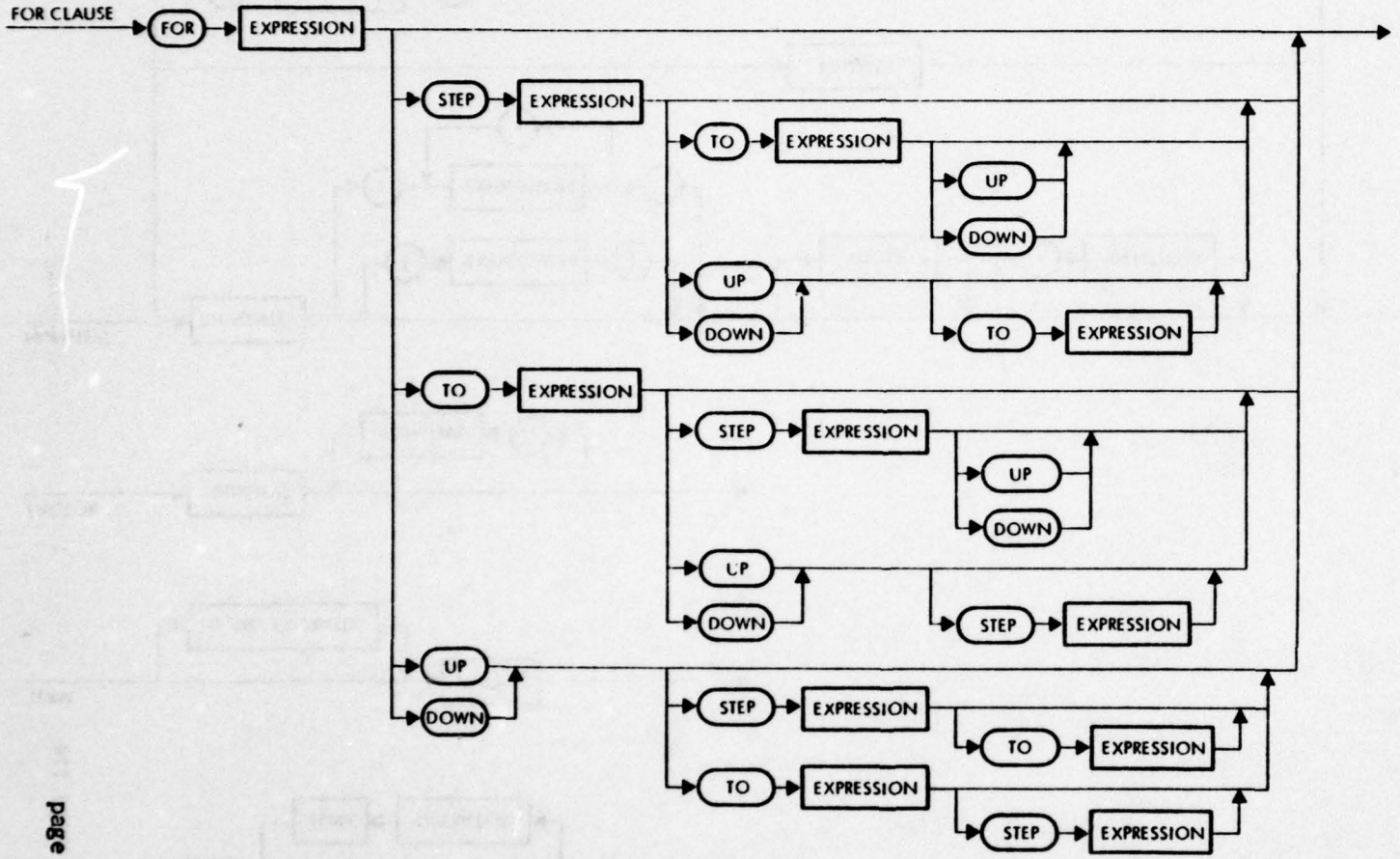


SMITE Syntax Diagrams

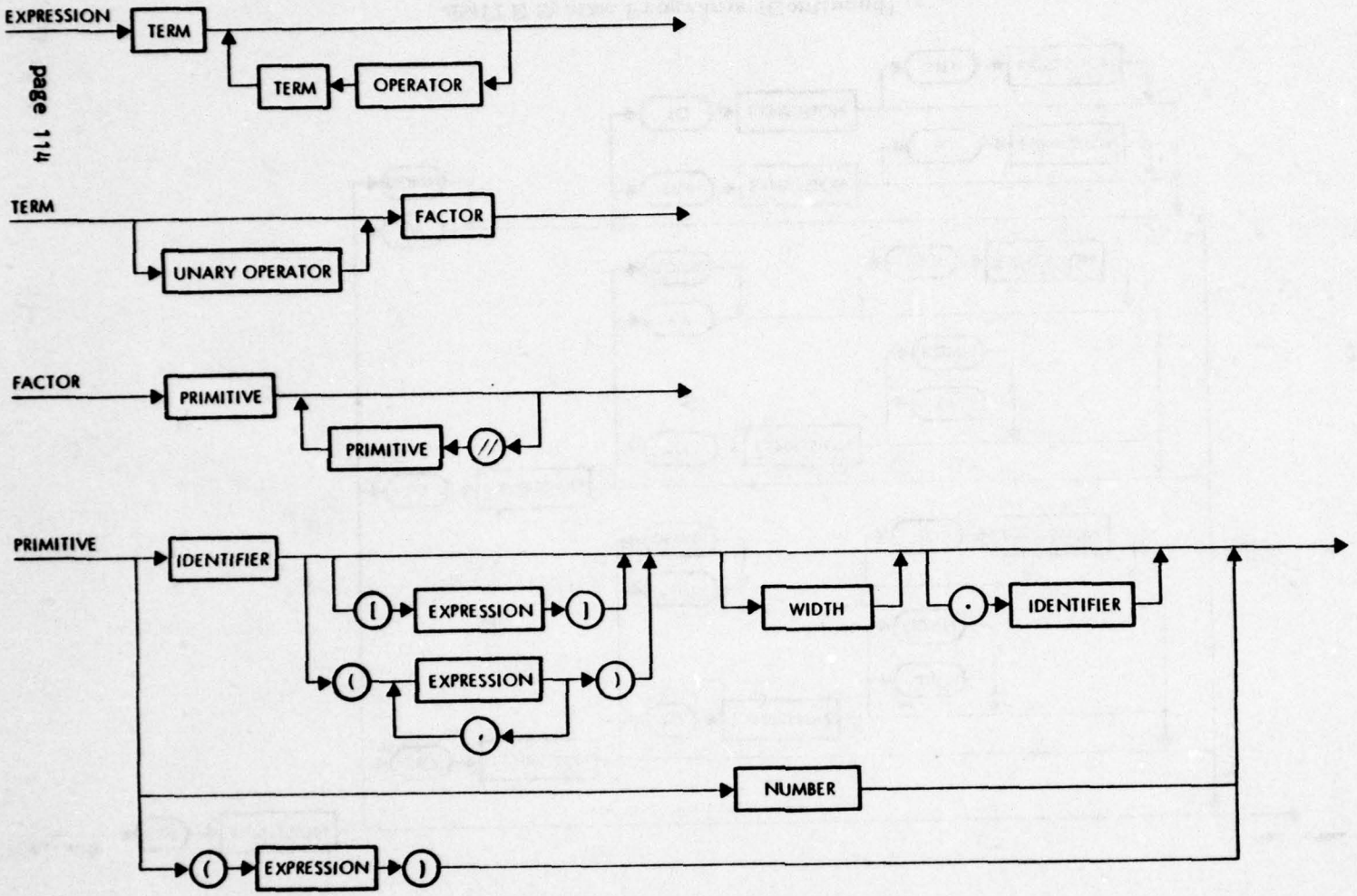


SMITE Syntax Diagrams (Continued)





SMITE Syntax Programs (Continued)



SMITE Syntax Diagrams (Concluded)

Appendix B: SMITE Compiler Operation Procedures

The SMITE compiler has been installed on the CDC-6000 computer at Kirtland Air Force Base, and is accessible over the ARPANET. After logging onto the SCOPE 3.4 Intercom system at Kirtland AFB, the following control cards may be used to execute the SMITE compiler:

```
ATTACH,SMITE,TRWSMITE,ID=TRWRADC.
RFL,200000.
SMITE,I=yourinput,L=youroutput,B=yourbinary.
```

The defaults for the input (I), output (L), and binary (B) files are INPUT, OUTPUT, and LGO, respectively. The defaults are used if the corresponding parameter is omitted.

Appendix C: Intel 8080 Microprocessor Definition

```
INTEL-8080: PROCESSOR;

    DECLARE DEFAULT<7:0> REGISTER,
        PROD-FLAGS<35:0>,
            STEP-FLAG FLAG DEFINED PROD-FLAGS<35>,
            INTERRUPT-IR<7:0> DEFINED PROD-FLAGS<25:18>,
            INTE FLAG DEFINED PROD-FLAGS<17>,
            INTERRUPT FLAG DEFINED PROD-FLAGS<16>,
           PC<15:0> DEFINED PROD-FLAGS<15:0>,
        REGS[0:3]<15:0>,
            REGISTERS[0:7]<7:0> DEFINED REGS,
                B DEFINED REGISTERS[0],
                C DEFINED REGISTERS[1],
                D DEFINED REGISTERS[2],
                E DEFINED REGISTERS[3],
                H DEFINED REGISTERS[4],
                L DEFINED REGISTERS[5],
                STATUS DEFINED REGISTERS[6],
                    CARRY FLAG DEFINED STATUS<0>,
                    PARITY FLAG DEFINED STATUS<2>,
                    AUX-CARRY FLAG DEFINED STATUS<4>,
                    ZERO FLAG DEFINED STATUS<6>,
                    SIGN FLAG DEFINED STATUS<7>,
                A DEFINED REGISTERS[7],
                    CARRY-A<8:0> DEFINED REGS[3]<8:0>,
        SP<15:0>,
        B-PAIR<15:0> DEFINED REGS[0],
        D-PAIR<15:0> DEFINED REGS[1],
        H-PAIR<15:0> DEFINED REGS[2],
        PSW<15:0> DEFINED REGS[3],
        IR,
            IR-OPERAND-SELECT<2:0> DEFINED IR<2:0>,
            IR-DEST-SELECT<2:0> DEFINED IR<5:3>,
            IR-SUB-FUNCTION<2:0> DEFINED IR<2:0>,
            IR-TEST-SELECT<2:0> DEFINED IR<5:3>,
        OP-REG,
        HOLD-A,
        OP-PAIR<15:0>,
        TEST FLAG,
        HOLD-REG,

        MEM[0:4095] MEMORY;
```

```
DECLARE
     OP-STEP EXTERNAL,
     OP-HALT EXTERNAL,
     OP-ERROR EXTERNAL,
     OP-NOTSIM EXTERNAL,
     IN-PORT PORT,
     OUT-PORT PORT;


'' ADD-REG ADDS AN OPERAND IN OP-REG TO THE REGISTER SELECTED BY''
'' IR-DEST-SELECT.                                              ''

ADD-REG: PROCESSOR;

     IF IR-DEST-SELECT = 6
          THEN IN 10 HOLD-REG <- MEM[H-PAIR] <-
                    MEM[H-PAIR] + OP-REG;
          ELSE IN 5 HOLD-REG <- REGISTERS[IR-DEST-SELECT] <-
                   REGISTERS[IR-DEST-SELECT] + OP-REG;
          END IF;
     SIGN <- HOLD-REG<7>;
     ZERO <- HOLD-REG = 0;
     PARITY <- HOLD-REG<7> XOR HOLD-REG<6> XOR HOLD-REG<5> XOR
               HOLD-REG<4> XOR HOLD-REG<3> XOR HOLD-REG<2> XOR
               HOLD-REG<1> XOR HOLD-REG<0> XOR 1;

     ADD-REG: END;

'' GETADD FETCHES A MEMORY ADDRESS WHICH IS LOCATED IN THE TWO  ''
'' BYTES IMMEDIATELY FOLLOWING THE CURRENT INSTRUCTION.  THE    ''
'' INSTRUCTION COUNTER *PC* IS UPDATED.  THE CLOCK IS NOT CHANGED''

GETADD: PROCESSOR;

     OP-PAIR <- MEM[PC+1] // MEM[PC];
     PC <- PC + 2;

     GETADD: END;

'' OPERAND FETCHES AN 8-BIT OPERAND BASED ON THE OPERAND SELECT  ''
'' BITS OF THE CURRENT INSTRUCTION.                             ''
```

```
OPERAND: PROCESSOR<7:0>;

     IF IR-OPERAND-SELECT /= 6
          THEN IN 1 OPERAND <- REGISTERS[IR-OPERAND-SELECT];
          ELSE IN 4 OPERAND <- MEM[H-PAIR];
          END IF;

     OPERAND: END;

'' PERFORM-ADD ADDS THE CONTENTS OF OP-REG TO THE ACCUMULATOR ''
'' AND SETS THE STATUS BITS.  THE CLOCK IS NOT ALTERED.          ''

PERFORM-ADD: PROCESSOR(CARRY-IN);

     DECLARE CARRY-IN FLAG;

     (AUX-CARRY // A<3:0>) <- CARRY-IN +
                              (0//A<3:0>) + OP-REG<3:0>;
     CARRY-A<8:4> <- AUX-CARRY +
                     (0//A<7:4>) + OP-REG<7:4>;
     SET-STATUS;

     PERFORM-ADD: END;

'' PERFORM-OP EXECUTES ARITHMETIC OPERATIONS BASED ON THE        ''
'' FUNCTION CODE IN IR.  THE CLOCK IS NOT ALTERED.  THE OPERANDS''
'' ARE *A* AND *OP-REG*.                                        ''

PERFORM-OP: PROCESSOR;

     CASE IR<5:3>;

          ADD: BEGIN;
               PERFORM-ADD(0);
               ADD: END;

          ADC: BEGIN;
               PERFORM-ADD(CARRY);
               ADC: END;

          SUB: BEGIN;
```

```
            OP-REG <- - OP-REG;
            PERFORM-ADD(0);
            CARRY <- NOT CARRY;
            SUB: END;

        SBB: BEGIN;
            OP-REG <- OP-REG;
            PERFORM-ADD( NOT CARRY );
            CARRY <- NOT CARRY;
            SBB: END;

        ANA: BEGIN;
            A <- A AND OP-REG;
            SET-STATUS;
            CARRY <- 0;
            ANA: END;

        XRA: BEGIN;
            A <- A XOR OP-REG;
            SET-STATUS;
            CARRY <- 0;
            XRA: END;

        ORA: BEGIN;
            A <- A OR OP-REG;
            SET-STATUS;
            CARRY <- 0;
            ORA: END;

        CMP: BEGIN;
            HOLD-A <- A;
            OP-REG <- - OP-REG;
            PERFORM-ADD(0);
            CARRY <- NOT CARRY;
            A <- HOLD-A;
            CMP: END;

        END CASE;

    PERFORM-OP: END;
'' POP REMOVES THE TOP TWO WORDS FROM THE CURRENT STACK AND      ''
```

```
'' RETURNS THEM AS THE PROCESSOR RESULT.                              ''

POP: PROCESSOR<15:0>;

     POP <- MEM[SP+1] // MEM[SP];
     SP <- SP + 2;

     POP: END;

'' PUSH STORES ITS OPERAND ONTO THE CURRENT STACK.  THE CLOCK IS ''
'' NOT ALTERED.                                                   ''

PUSH: PROCESSOR(VALUE);

     DECLARE VALUE<15:0>;

     SP <- SP - 2;
     (MEM[SP+1] // MEM[SP]) <- VALUE;

     PUSH: END;

'' SET-TEST-STATUS EVALUATES TEST CONDITIONS FOR CONDITIONAL      ''
'' JUMPS, CALLS, AND RETURNS.  IF THE CONDITION IS TRUE THEN THE ''
'' FLAG *TEST* WILL BE SET TRUE.  THE CLOCK IS NOT ALTERED.       ''

SET-TEST-STATUS: PROCESSOR;

     CASE IR-TEST-SELECT;

          '' NZ ''  TEST <- NOT ZERO;

          '' Z  ''  TEST <- ZERO;

          '' NC ''  TEST <- NOT CARRY;

          '' C  ''  TEST <- CARRY;

          '' PO ''  TEST <- NOT PARITY;

          '' PE ''  TEST <- PARITY;

          '' P  ''  TEST <- NOT SIGN;
```

```
          '' M  ''  TEST <- SIGN;

          END CASE;

     SET-TEST-STATUS: END;

'' SET-STATUS INTERPRETS THE CURRENT ACCUMULATOR CONTENTS TO SET''
'' THE SIGN, ZERO, AND PARITY BITS IN THE PSW.                  ''
'' THE CLOCK IS NOT ALTERED.                                    ''

SET-STATUS: PROCESSOR;

     SIGN <- A<7>;
     ZERO <- A = 0;
     PARITY <- A<7> XOR A<6> XOR A<5> XOR A<4> XOR
               A<3> XOR A<2> XOR A<1> XOR A<0> XOR 1;

     SET-STATUS: END;

'' STORE STORES A VALUE BASED ON THE DESTINATION SELECT BITS.   ''
'' THE CLOCK IS UPDATED.                                        ''

STORE: PROCESSOR(VALUE);

     DECLARE VALUE;

     IF IR-DEST-SELECT /= 6
          THEN IN 3 REGISTERS[IR-DEST-SELECT] <- VALUE;
          ELSE IN 6 MEM[H-PAIR] <- VALUE;
          END IF;

     STORE: END;

'' ************************************************************** ''
'' ************************************************************** ''
''                                                                ''
''                          MAIN PROCESSOR                        ''
''                                                                ''
'' ************************************************************** ''
'' ************************************************************** ''
```

```
'' INITIALIZATION.                                                    ''

PC <- 0;
STATUS <- 2;
INTE <- INTERRUPT <- 0;

DO FOREVER; BEGIN;


'' FIRST OFF, CHECK IF PROD HAS SET THE EMULATOR STEP FLAG, AND       ''
'' DROP TO PROD IF SO.                                               ''

IF STEP-FLAG
     THEN BEGIN;
          OP-STEP;
          END;
     END IF;

'' FETCH INSTRUCTION.                                                ''

IF INTERRUPT AND INTE
     THEN BEGIN;
          INTERRUPT <- INTE <- 0;
          IR <- INTERRUPT-IR;
          END;
     ELSE BEGIN;
          IR <- MEM[PC];
          PC <- PC + 1;
          END;
     END IF;

'' DECODE ON THE UPPER TWO BITS.                                     ''

CASE IR<7:6>;

     '' 00 -- MISCELLANEOUS INSTRUCTIONS.                            ''

     CODE00: CASE IR-SUB-FUNCTION;

          '' NOP '' IN 4  1;
```

```
'' LXI AND DAD'' IF IR<3>
        THEN IN 10 DAD: BEGIN;
             IF IR<5:4> /= 3
                  THEN OP-PAIR <- REGS[IR<5:4>];
                  ELSE OP-PAIR <- SP;
                  END IF;
             (CARRY // H-PAIR) <- OP-PAIR + 0//H-PAIR;
             DAD: END;
        ELSE IN 10 LXI: BEGIN;
             GETADD;
             IF IR<5:4> /= 3
                  THEN REGS[IR<5:4>] <- OP-PAIR;
                  ELSE SP <- OP-PAIR;
                  END IF;
             LXI: END;
        END IF;

'' STA, LDA, LDAX, STAX, SHLD, LHLD '' CASE IR<5:3>;

     STAX-B: IN 7 MEM[B-PAIR] <- A;

     LDAX-B: IN 7 A <- MEM[B-PAIR];

     STAX-D: IN 7 MEM[D-PAIR] <- A;

     LDAX-D: IN 7 A <- MEM[D-PAIR];

     SHLD: IN 16 BEGIN;
          GETADD;
          MEM[OP-PAIR] <- L;
          MEM[OP-PAIR + 1] <- H;
          END;

     LHLD: IN 16 BEGIN;
          GETADD;
          L <- MEM[OP-PAIR];
          H <- MEM[OP-PAIR + 1];
          END;

     STA: IN 13 BEGIN;
          GETADD;
```

```
            MEM[OP-PAIR] <- A;
            END;

      LDA: IN 13 BEGIN;
            GETADD;
            A <- MEM[OP-PAIR];
            END;

      END CASE;

'' INX, DCX '' IN 5 BEGIN;
      IF IR<3>
            THEN OP-PAIR <- -1;
            ELSE OP-PAIR <- 1;
            END IF;
      IF IR<5:4> /= 3
            THEN REGS[IR<5:4>] <- REGS[IR<5:4>] + OP-PAIR;
            ELSE SP <- SP + OP-PAIR;
            END IF;
      END;

INR: BEGIN;
      OP-REG <- 1;
      ADD-REG;
      END;

DCR: BEGIN;
      OP-REG <- -1;
      ADD-REG;
      END;

MVI: BEGIN;
      OP-REG <- MEM[PC];
      PC <- PC + 1;
      IF IR-DEST-SELECT /= 6
            THEN IN 7 REGISTERS[IR-DEST-SELECT] <- OP-REG;
            ELSE IN 10 MEM[H-PAIR] <- OP-REG;
            END IF;
MVI: END;

'' ROTATES, CMA, CARRYS, DAA '' IN 4 CASE IR<5:3>;
```

```
          RLC: BEGIN;
               CARRY <- A<7>;
               A <- SLC(A,1);
               RLC: END;

          RRC: BEGIN;
               CARRY <- A<0>;
               A <- SRC(A,1);
               RRC: END;

          '' RAL'' CARRY-A <- SLC(CARRY-A, 1);

          '' RAR '' CARRY-A <- SRC(CARRY-A, 1);

          DAA: BEGIN;
               IF (A<3:0> > 9)  OR  AUX-CARRY
                    THEN (AUX-CARRY//A<3:0>) <-
                              (0//A<3:0>) + 6;
                    END IF;
               IF (A<7:4> > 9)  OR  CARRY
                    THEN CARRY-A <- (0//A) + X'60';
                    END IF;
               SET-STATUS;
               DAA: END;

          '' CMA '' A <- NOT A;

          '' STC '' CARRY <- 1;

          '' CMC '' CARRY <- NOT CARRY;

          END CASE;

     CODE00: END CASE;

'' 01 -- DATA TRANSFER INSTRUCTIONS.                              ''

CODE01: IF IR = X'76'
     THEN IN 7 OP-HALT;
     ELSE STORE(OPERAND);
     CODE01: END IF;
```

```
'' 10 -- ACCUMULATOR FUNCTIONS.                                    ''

CODE10: BEGIN;
     OP-REG <- OPERAND;
     IN 3 PERFORM-OP;
     CODE10: END;


'' 11 -- TRANSFERS AND OTHER STUFF.                                ''


CODE11: CASE IR-SUB-FUNCTION;

     '' 000 - CONDITIONAL SUBROUTINE RETURNS.                      ''

     BEGIN;
          IN 5 SET-TEST-STATUS;
          IF TEST
               THEN IN 6 PC <- POP;
               END IF;
          END;

     '' 001 - RETURN, POP, SPHL, PCHL.                             ''

     CASE IR<5:3>;

          POP-B: IN 10 B-PAIR <- POP;

          RETURN: IN 10 PC <- POP;

          POP-D: IN 10 D-PAIR <- POP;

          '' ILLEGAL OPCODE '' OP-NOTSIM;

          POP-H: IN 10 H-PAIR <- POP;

          PCHL: IN 5 PC <- H-PAIR;

          POP-PSW: IN 10 PSW <- POP;
```

```
     SPHL: IN 5 SP <- H-PAIR;

     END CASE;

'' 010 - CONDITIONAL JUMPS.                                    ''

IN 10 BEGIN;
     GETADD;
     SET-TEST-STATUS;
     IF TEST
          THEN PC <- OP-PAIR;
          END IF;
     END;

'' 011 - MISCELLANEOUS.                                        ''

CASE IR<5:3>;

     JMP: IN 10 BEGIN;
          GETADD;
          PC <- OP-PAIR;
          END;

     '' ILLEGAL OPCODE '' OP-NOTSIM;

     OUT: IN 10 BEGIN;
          IR <- MEM[PC];
          OUT-PORT <- A;
          PC <- PC + 1;
          END;

     INPUT: IN 10 BEGIN;
          IR <- MEM[PC];
          A <- IN-PORT;
          PC <- PC + 1;
          END;

     XTHL: IN 18 BEGIN;
          OP-PAIR <- MEM[SP+1] // MEM[SP];
          (MEM[SP+1] // MEM[SP]) <- H-PAIR;
          H-PAIR <- OP-PAIR;
```

```
          END;

     XCHG: IN 4 BEGIN;
          OP-PAIR <- H-PAIR;
          H-PAIR <- D-PAIR;
          D-PAIR <- OP-PAIR;
          END;

     DI: IN 4 INTE <- 0;

     EI: IN 4 INTE <- 1;

     END CASE;

'' 100 - CONDITIONAL CALLS.                                    ''

BEGIN;
     IN 11 BEGIN;
          SET-TEST-STATUS;
          GETADD;
          END;
     IF TEST
          THEN IN 6 BEGIN;
               PUSH(PC);
               PC <- OP-PAIR;
               END;
          END IF;
     END;

'' 101 - CALL AND PUSH.                                        ''

IF IR<3>
     THEN IF IR<5:4> = 0
               THEN CALL: IN 17 BEGIN;
                    GETADD;
                    PUSH(PC);
                    PC <- OP-PAIR;
                    END;
               ELSE '' ILLEGAL OPCODE '' OP-NOTSIM;
               END IF;
     ELSE PUSH-INSTR: IN 11 PUSH(REGS[IR<5:4>]);
     END IF;
```

```
            '' 110 - IMMEDIATE OPERAND ARITHMETIC.                    ''

            IN 7 BEGIN;
                 OP-REG <- MEM[PC];
                 PC <- PC + 1;
                 PERFORM-OP;
                 END;

            '' 111 - RST                                              ''

            RST: IN 11 BEGIN;
                 PUSH(PC);
                 PC <- 0;
                 PC<5:3> <- IR<5:3>;
                 END;

            CODE11: END CASE;

       END CASE;

END;  INTEL-8080: END;  *EOR *EOR,17
```

Appendix D: Augmented MULTI Micromachine Definition

Four microinstructions have been added to the standard MULTI microinstruction set for use by the SMITE compiler. These new microinstructions are as follows:

MPY A,B

The positive quantity in register A is multiplied by the positive quantity in register B to produce a positive quantity in registers A//(A+1).

DIV A,B

The positive quantity in registers A//(A+1) is divided by the positive quantity in register B to produce a positive quotient in register A+1, and a remainder in register A.

SHFTX A,B,CDE

The SHFTX microinstruction functions exactly like the standard SHIFT microinstruction except that the parameter E is a register pointer rather than a shift count, and that on double length right arithmetic shifts, the carry bit is not shifted in (a true 36-bit shift is performed).

STMSK A,B

The value in register A is stored under the mask in register B into the main store location addressed by R.MX. R.MX is incremented by one after the store. The stored value is (A AND B) OR (M AND NOT B).

In addition, the operation of two of the Nanodata standard microinstructions has been modified as follows:

SHIFT A,B,CDE

The SHIFT instruction does not perform a 37-bit operation involving the carry bit for double right arithmetic shifts. Instead, a true 36-bit shift is performed. (*Note: the SHIFT* instruction has been redefined using another opcode. Both shifts are therefore available.)

ALUX A,B,CDE

The status returned in FIST by the ALUX instruction has been modified to change the definition of zero status. Zero is now returned if and only if the ALUX operation generated a zero result, and zero status was set on input to the ALUX instruction.

Appendix E: SMITE Compiler Version 1.0 Error Messages

Numerous error diagnostics are built into the SMITE compiler. These diagnostics generate error messages when violations of the SMITE language definition or of limits of the SMITE compiler are violated.

The error messages are of several forms. Errors detected while scanning the input stream are printed interspersed with the listing of the input program, with an arrow provided to indicate the approximate location of the input scan when the error was detected. Errors detected during the storage allocation phase of the compiler are printed above the allocation data line for the offending item. Errors detected during code generation are printed above a reconstruction of the statement being processed when the error was detected, although compound statements (e.g. 'BEGIN' .. 'END') can interfere with the ordering of the printout. Errors detected during the second storage allocation phase include the processor in which the error was detected. Errors detected in the assembly phase are printed either immediately before the assembly listing and identify the offending value, or else are printed on the same line as the offending line of microcode.

72 BIT SHIFT NOT IMPLEMENTED

No microinstructions are presently available for shifting operands larger than 36 bits, and so shifts on large operands are not implemented in SMITE.

ALLOCATION IMPOSSIBLE

This error results when a proper register allocation request is made in the compiler but the registers are unavailable. If the problem persists after correcting all other errors, try reducing the nesting level of DO and/or IN statements. If the problem cannot be corrected in this manner, save the input and output for analysis.

AMBIGUOUS PROCESSOR REFERENCE

Two external references (i.e. data items declared as PORT or EXTERNAL) of the same name have been declared in separate scopes. Both references are mapped to the same external linkage.

ATTRIBUTE ILLEGAL FOR FORPAR

The compiler has detected a formal processor parameter declared as either EXTERNAL or DATA. Both forms are illegal in SMITE.

BAD ARGUMENT LIST

The closing parenthesis for an actual parameter list in a processor reference was not found. Text is skipped to the right parenthesis or a semicolon.

BAD ATTRIBUTE CHAIN

The compiler operand tables have been destroyed. If the error persists after correcting all other errors, save the input and output for analysis.

BAD ATTS/RTA

The compiler operand tables have been destroyed. If the error persists after correcting all other errors, save the input and output for analysis.

BAD BUILTIN FUNCTION CALL

An incorrect number of parameters were supplied in a call to one of the builtin functions (e.g. SE, SLC, etc.)

BAD DIMENSION LIST

The closing bracket (i.e. ']') for an array declaration or array reference was not found when expected after the last number (or possibly identifier, for an array reference), or else a number was not found after a colon within the list. Text is skipped to a bracket or semicolon.

BAD DIMENSIONS

The lower bound in an array declaration (i.e. '[number:number]') was greater than the upper bound. The two numbers are interchanged.

BAD IDENTIFIER SYNTAX

An identifier was found with a hyphen as the last character. The identifier is accepted. Identifiers ending in a hyphen are illegal in SMITE because of the possible confusion with text such as 'A- B' and 'A - B'.

BAD PARAMETER LIST

No terminating parenthesis was found when expected to close the parameter list definition. Text is skipped to a left parenthesis or semicolon.

BAD REGISTER DROP

An error has occurred causing the compiler to attempt to release registers not owned by the releasing table entry. If the error persists after correcting all other errors, save the input and output for analysis.

BLOCK MAY NOT BE MOVED TO TEMP

An attempt has been made to move a register which is not to be altered to temporary storage. If the error persists after correcting other errors and simplifying the nesting of IN and DO statements, save the input and output for analysis.

CHANGE TO LOCKED REGISTER

A non-destructive transform (e.g. move or shift) has been made to a register which was not to be altered. If the error persists after correcting all other errors, save the input and output for analysis.

COMPILER ERROR

An improper translation of a processor call expression has been made, or improper symbol table information has been detected trying to create a symbol table entry. Previous syntax errors in the SMITE description may cause this error. If the error persists after correcting all previous syntax errors, save all compiler input and listings for analysis.

CONCATENATE EXPRESSION TOO WIDE

The result of concatenating two operands was greater than 72 bits, exceeding the capacity of the compiler.

CONTAINER NOT EXPANDED

The widening of an operand required the allocation of registers in addition to those already allocated to the operand, which is not presently implemented. Save the input and output for analysis.

CONTAINER TOO SMALL

The compiler has tried to allocate a group of registers too small to hold the corresponding operand. If the error persists after correcting all other errors, save the input and output for analysis.

CONTAINER TRUNCATION

An operand was moved by the compiler to a register block too small to hold the operand. Save the input and output for analysis.

DEFAULT MAY NOT BE EXTERNAL

The default data declaration may not have the storage class attribute EXTERNAL. Remove the EXTERNAL attribute.

DEFINED ITEM DIRECTION INVALID

A width was specified in a DEFINED attribute, but the bit numbering scheme is not compatible with the bit numbering of the parent. The two bit numbers in the DEFINED width are reversed.

DEFINED LENGTH PHRASE INVALID

One of the subscripts in the length phrase of this subfield data item DEFINED attribute is out of range of the addressing of the parent. Allocation is terminated for this data item.

DEFINED WIDTH INVALID

The width specified in this data item DEFINED attribute could not

be mapped onto the parent, either because the widths were not equal or because one of the specified parent bits is out of range of the parent definition.

DEFINED WIDTH PHRASE DELETED

In order for the subfield definition to conform to the parent, the width in a DEFINED attribute covering multiple parent elements must equal that of the parent. This data item violates that rule. The offending subfield width is deleted from the DEFINED attributes, and the width of the parent is substituted.

DEFINITION PHRASE ILLEGAL

Data items declared as EXTERNAL and the DEFAULT item may not be declared as DEFINED on another item.

DEGENERATE PARALLEL CONTEXT

A parallel context has been found which contains no statements.

DIMENSIONED TEMPLATE INVALID

An array declaration is illegal for a DATA type item, and is ignored.

DUPLICATE CONTEXT LABEL

The label of the context block being opened duplicates that of a context elsewhere in the current processor. The label is ignored.

DUPLICATE DATA-ITEM DEFINITION

The identifier in this declaration has been previously declared within the current processor. The declaration of the item is ignored and text skipped to a comma or semicolon.

DUPLICATE LABEL GENERATED

Multiple identical labels have been output by the compiler in the generated microcode. Save the input and output for analysis.

DUPLICATE PARAMETER

The same data item name appears twice in the parameter list for a processor. The latter parameter definition is accepted for purposes of error analysis of parameter lists in calls on the processor, but it is dissociated from the declared name.

DUPLICATE PROCESSOR NAME

A processor of the same name has been previously declared in the scope of the current processor. This definition is syntax checked, but is dissociated from the processor name. No code will be generated for this processor definition.

END MISSING BUT ASSUMED

An END statement (e.g. END, END IF, etc.) for the current context block was not found before an END for a surrounding context, but has been assumed by the compiler. Insert the required statement.

ESCAPE MAY DESTROY REGISTERS

An ESCAPE statement transfers control past the context block controlling a DO statement. The resulting microcode may not restore registers or temporaries properly, and therefore may not execute properly. Examine the microcode for the DO loop output by the compiler, and verify proper handling of registers. If incorrect microcode is generated, recode the loop to avoid the ESCAPE past the scope of the DO.

FUNCTION DOES NOT HAVE VALUE

The processor name in a function processor was never assigned a value (i.e. never appeared as the left operand of a store ('<-') operator).

FUNCTION RESULT WIDTH TOO LARGE

The function result of a processor was declared as more than 72 bits. The width is truncated to 72 bits.

FUNCTION WIDTH UNDEFINED

The current processor was used as a function before its definition, but the definition provided was not of a function processor. A 72-bit result is assumed.

ILLEGAL ALLOCATION REQUEST

The compiler has attempted to allocate a register block greater than 72 bits wide, has attempted to allocate a register block not 1, 2, or 4 registers wide, has attempted to allocate a register block outside the available register address space, or has attempted to allocate a multi-register block not on an even register address boundary. If the error persists after correcting all other errors, save the input and output for analysis.

ILLEGAL ARRAY DEFINITION

A data item declared as EXTERNAL, CLOCK or DATA may not be defined to be an array. Remove the array definition.

ILLEGAL ARRAY PARAMETER

This error should never occur in the Version 1 SMITE compiler. If it does, save the input and output for analysis.

ILLEGAL DATA-ITEM NAME

The identifier being declared or the parent data item identifier in a DEFINED phrase was not a proper data item name. The DEFINED phrase was ignored, if the error occurred in a DEFINED phrase, and text skipped to a comma or semicolon.

ILLEGAL DECLARATION OF RESERVED WORD

A parameter in the parameter list of a processor header or the identifier in a declaration is one of the SMITE reserved words. The parameter is ignored. Change the parameter name.

ILLEGAL NUMBER

The closing quote was not found in a binary, octal, or hexadecimal number. Text is skipped to the closing quote or a semicolon.

ILLEGAL PARAMETER

A parameter in a processor declaration was found to be other than an identifier. The parameter is ignored.

ILLEGAL PARAMETER CLASS

The storage class attribute of the formal parameter is not compatible with the actual parameter storage attribute (if any).

ILLEGAL PROCESSOR REFERENCE

A processor reference with parameters was found on the left hand side of a store operation. The only form of processor name reference allowed on the left hand side of an assignment is the name itself, without parameters, which signifies that the function processor is being assigned a value.

ILLEGAL RANGE FOR ESCAPE

An ESCAPE statement results in transfer of control past a timing or parallel context block. Recode the description to conform to the SMITE language requirements for interaction of ESCAPE statements with timing and parallel context blocks.

ILLEGAL SET PARAMETER

A formal parameter is stored into (i.e. set) by the called processor, and the corresponding actual parameter is a computed expression (e.g. 'A + B', a constant, a function call etc.).

ILLEGAL STORE OPERAND

An expression was found which attempts to store into a processor reference, a computed expression, a constant, or a switch. The expression is deleted.

IMPOSSIBLE DEALLOCATION REQUEST

The compiler has attempted to free a block of registers which are being maintained in temporary storage. If the error persists after correcting all other errors, save the input and output for analysis.

INCORRECT END LABEL IGNORED

An END statement (e.g. END, END IF, etc.) for the current context block was found but has the wrong label. The label was ignored and the statement accepted by the compiler.

INCORRECT END STATEMENT ACCEPTED

An END statement (e.g. END, END IF) of the wrong type (e.g. END instead of END CASE) for the current context block was found by the compiler. The statement was assumed of the correct type by the compiler.

INCORRECT NUMBER OF SUBFIELD ELEMENTS

A partial word of the parent is required to map the subfield data item onto the parent. Allocation is terminated for this data item.

INVALID ALLOCATION REQUEST

A request for a block of 3 registers was made in the code generator. If the error persists after correcting all other errors, save the input and output for analysis.

INVALID ARRAY DECLARATION

An array declaration with only one address element was found.

Both a lower and upper bound is required for array addressing definition.

INVALID DEBUG COMMAND

An improperly formed debug command has been encountered; a number was expected in an item of the form "identifier + number," but no number was found. Debug commands are provided only for use in debugging the operation of the SMITE compiler, and are intended for use by SMITE compiler programmers.

INVALID DEFINED LENGTH PHRASE DELETED

A length phrase was found in the DEFINED attribute for the current data item, but the parent data item is not an array.

INVALID DEFINED SUBSCRIPT

A single subscript referencing a parent was present in the DEFINED attribute for this data item, and was out of range of the array addresses of the parent. The subscript is reset to the corresponding address limit.

INVALID DO STATEMENT

The termination clause in a DO statement was neither FOREVER, WHILE, UNTIL, or FOR. Text is skipped to a semicolon. No looping will occur.

INVALID EXTRACT OPERAND

The subfield specified for extraction via a width phrase (i.e. '<number:number>') or a data item template (i.e. declared as DATA) has bit numbering incompatible with the parent operand.

INVALID REQUESTED RESIDENCE

The compiler has attempted to allocate registers when not required to. If the error persists after correcting all other errors, save the input and output for analysis.

INVALID SUBFIELD REFERENCE

In a data item reference of the form ITEM.DATA, either no DATA identifier was found, or else the identifier that was found did not have the DATA attribute. The expression is deleted.

INVALID SUBSCRIPT

A subscript was found on a data item not declared as an array. The subscript is ignored.

INVALID USE OF DATA TEMPLATE

An identifier found as part of an expression was declared with the DATA attribute, and is therefore restricted to uses such as 'ITEM.DATA'.

INVALID WIDTH DECLARATION

The width declaration for this identifier could not be processed, and is ignored. Text is skipped to a right angle bracket ('>') or a semicolon.

LABEL FOUND IN EXPRESSION

An identifier found as part of an expression has been previously

encountered as a context label. The current expression is deleted.

LABEL MISSING ON END BUT ASSUMED

An END statement (e.g. END, END IF) for the current context block was found but did not have a required context label. The label was assumed by the compiler.

LABEL NOT ALLOWED

A label was found on a statement for which no label is allowed (i.e. expressions or ESCAPE). A BEGIN statement associated with the label is assumed. Thus

```
LABEL: ESCAPE;
```

will be translated by the compiler to

```
LABEL: BEGIN;

   ESCAPE;
```

In this particular example, the statement will act as a no-operation statement, since the ESCAPE will transfer control to the end of the context block created to surround it.

LENGTH AND WIDTH INCONSISTENT

The subfield data item mapping onto the parent does not cover all the bits of each specified parent word. Allocation is terminated for this subfield data item.

LENGTH DECLARATION MISSING

No valid array bounds definition (i.e. '[number:number]') was

found in the computer description, however the compiler has determined that one is required.

LIGHT CANNOT BE READ

A data item declared as LIGHT (write only storage) is used as a read operand in an expression.

MISMATCHED PARENTHESES

The closing parenthesis for a parenthesized expression was not found when expected. Text is skipped until the required right parenthesis or a semicolon is found.

MISPLACED END STATEMENT

An END statement (e.g. END, END IF) was found when a statement (e.g. an expression, IF, DO, CASE, etc.) was required for a context block.

MISPLACED KEYWORD

An identifier found as part of an expression is a SMITE reserved word and therefore cannot be part of an expression. The expression is deleted by the compiler.

MISSING KEYWORD

The keyword 'THEN' was not found when expected in an IF statement. Text was skipped until 'THEN' or a semicolon was found. If 'THEN' was found, processing of the IF statement was continued. Otherwise the statement processing stops and the statement is deleted.

MISSING OR MISPLACED]

The closing right square bracket in a subscript was not found. Text is skipped to the bracket or a semicolon.

MULTIPLE CLOCK DEFINITION

Multiple data items have been declared as the clock. SMITE only permits one such declaration.

NO ARGUMENTS

No arguments could be found within a parameter list in a processor header, although the parenthesis delimiting the start of such a list was found. Compilation continues normally.

NO REGISTER ALLOCATED

The code generator was not able to allocate a register when required. This error occurs due to too many processor parameters or too complex an expression.

NO STORAGE CLASS

No attribute for storage class (e.g. REGISTER, PORT, etc.) has been defined for the data item. A storage class attribute may be defined by explicit declaration, by inheritance from a parent, or by default. REGISTER is assumed.

NOT ALL INPUT PROCESSED

The compiler has closed the main processor definition at an 'END' statement, but more text remained in the input. This error may

be due to improper context block nesting such as extra END statements. When performing error recovery for context block nesting problems, the compiler never deletes an 'END' statement, and may on occasion add one. This recovery action can also lead to this error.

NOT ENOUGH PARAMETERS

More formal parameters were declared than were coded in the actual processor call.

ONESPAD SE MODE NOT IMPLEMENTED

This error should never occur in the Version 1 SMITE compiler. If it does, save the input and output for analysis.

PARALLEL DONE SERIALLY

The PARALLEL-BEGIN and PARALLEL-END statements are implemented as BEGIN and END, respectively. Verify that the SMITE input will operate properly if the statements in parallel context are executed serially.

PARENT/SUBFIELD CLASSES INCOMPATIBLE

The storage class attributes of the parent and subfield must be identical with the exception that memory and REGISTER are considered equivalent for this purpose. The subfield is assumed to have the same storage class attribute as the parent.

PROBABLE INFINITE LOOP

A DO FOREVER statement was encountered which was never ESCAPEd in

order to terminate the loop. This error is normal for most main instruction interpretation loops.

PROCESSOR MUST BE FUNCTION

An expression uses a processor call as a function, but the processor was not defined to return a value.

PROCESSOR W/NO WIDTH USED IN EXTRACT

Due to the operation of the storage allocation part of the compiler, this error should never occur. If it does, save the input and output for analysis.

RECURSION ILLEGAL

A recursive processor call has been detected, and is not allowed in SMITE.

RECURSIVE CALL

A recursive processor call has been found, which is not permitted in SMITE.

REFERENCE TO UNDECLARED DATA-ITEM

An identifier found as part of an expression was never declared and has been determined by the compiler not to be an as-yet undeclared processor. SMITE requires that all data items used in a computer description be explicitly declared in the scope of their use.

REG BLOCK ILLEGALLY LOCKED

While attempting to store all active registers into temporaries in preparation for a processor call, the code generator encountered a group of registers locked against any alteration. If this error persists after correcting all other errors, save the input and output for analysis.

REG TREE NOT REGISTER

A compiler error has occurred in handling temporary variables for loop control, addressing, etc. Save the input and output for analysis.

REQUESTED WIDTH LESS THAN ACTUAL WIDTH

This error results when an expression is evaluated and has a width greater than required. It results, for example, when a value is stored into a data item smaller than the expression, when an expression is wider than its corresponding formal parameter in a processor call, or when the result of evaluating the selector expression in an IF statement is greater than one bit wide.

SEMICOLON MISSING - TEXT SKIPPED

No semicolon was found at the end of an expression. Text is skipped until a semicolon or 'END' is found.

SEMICOLON MISSING BUT ASSUMED

A semicolon was expected to terminate the current statement, but was not found. One is assumed.

SEMICOLON OR COMMA MISSING

No terminator for the current data item declaration was found. Text is skipped until a comma or semi-colon is found.

SUBFIELD TOO WIDE

The data item is a subfield of a parent data item, but the declared width of the item is greater than the width of the parent's subfield.

SUBFIELD/PARENT WIDTH DISCREPANCY

No subfield width was stated in the DEFINED clause for this data item, and so the entire parent width was used for allocation. The width of the parent item is greater than the width of the declared subfield data item, however, and therefore the two data items cannot be reconciled.

SUBSCRIPT MISSING

No subscript was found following a data item reference for an array. The expression is deleted.

SUBSCRIPT OUT OF RANGE

A constant subscript in an array reference is out of the bounds of the address space declared for the array.

TEMP DESTROYED

The compiler has attempted to restore information from temporary storage to registers, but has discovered that the data has been

altered during storage. If the error persists after correcting all other errors, save the input and output for analysis.

TEMP USED BEFORE BEING SET

A temporary operand was referenced before being assigned a value. Save the input and output for analysis.

TOO FEW CASE SUBSTATEMENTS

The CASE statement requires 2**N substatements where N is the width of the selector expression. Not enough statements were supplied; NULL was provided for the missing statements.

TOO MANY CASE SUBSTATEMENTS

The CASE statement requires 2**N substatements, where N is the width of the selector expression. Too many were supplied; the extras are ignored.

TOO MANY PARAMETERS

More actual parameters were coded in the SMITE input than defined for the called processor.

UNABLE TO RESTORE TEMP

No register for restoration of an operand from temporary storage could be obtained. Save the input and output for analysis.

UNDECLARED DATA ITEM NOT ALLOCATED

A data item, such as a formal parameter or the parent in a DEFINED clause, was never declared. No storage allocation is made for the data item or its subfields, if any.

UNDECLARED ITEMS -- NOT ALLOCATED

Data items were declared in the current processor which were not allocated by the compiler due to circularity in their DEFINED attributes. Rewrite the DEFINED attributes so that all DEFINED items ultimately have a non-DEFINED parent.

UNDECLARED PROCESSOR(S)

Processor references were found in the computer description which were never matched by corresponding processor declarations. This error can also be due to undeclared variables which were incorrectly assumed to be processor references by the compiler.

UNKNOWN LABEL FOR ESCAPE

The target label in an ESCAPE statement could not be found in the current processor. The statement is deleted.

UNKNOWN LABEL ON END IGNORED

An END statement (e.g. END, END IF) was found having a context label for a context block currently unknown to the compiler as an open context block. The label was ignored by the compiler.

UNRECOGNIZABLE KEYWORD

The compiler was attempting to process a statement of the form END IF or END CASE but the keyword following the END keyword was

unrecognizable. The compiler ignored the bad keyword and skipped text to the next semicolon.

UNRECOGNIZABLE OPERAND

An expression operand (i.e. a constant, identifier, or parenthesized expression) could not be found, but the syntax up to this point in the current expression required one to follow. The current expression is deleted.

UNRECOGNIZED ATTRIBUTE IGNORED

An attribute other than REGISTER, MEMORY, LIGHT, SWITCH, DATA, PORT, EXTERNAL, or CLOCK, has been found. No storage class is set for this item, which may cause allocation errors.

UNRECOGNIZED EXPRESSION TREE

A bad input has been given to the code generator. If the error persists after correcting all other errors, save the input and output for analysis.

UNRECOGNIZED STORE OPERAND

The operand on the left hand side of the store operator could not be interpreted. This error should never occur in the Version 1 SMITE compiler. If it does, save the input and output for analysis.

WIDTH AND FLAG CONFLICT

A width declaration (i.e. '< ... >') has been given for a data item declared as a FLAG (a one-bit register). Remove the width declaration, or change the attribute to REGISTER.

WIDTH LIMIT EXCEEDED

The data item has been declared with a total width of greater than 72 bits. The SMITE compiler is unable to handle items wider than 72 bits; the computer description will have to be restructured to permit compilation.

WIDTH NOT DEFINED

No width declaration could be determined for this data item, either from explicit declaration, inheritance from a parent data item, or default declaration. A width of '<1:72>' is assumed.

WIDTH/MODE INCOMPATIBILITY

If this error occurs, the compiler has improperly processed a conditional jump. If the error persists after correcting all other errors, save the input and output for analysis.

Appendix F: FTSC Emulator Requirements Specification

F-1.0 SCOPE

This specification defines the requirements for the development of the Fault-Tolerant Spaceborne Computer (FTSC) emulator. A basic emulation of the Fault-Tolerant Spaceborne Computer will be developed at the simplex processor instruction level to provide bit-identical results except where code execution results are time dependent or where fault-tolerant features affect execution. The basic FTSC emulation will be implemented using a higher-order programming language or a macro-structured assembly language technique to provide easy separation of critical functions for later enhancement and for compatibility with both 18-bit and 36-bit versions of Nanodata equipment used to support the emulation. The FTSC emulation will be tested to demonstrate that the requirements have been met and to measure the specific performance achieved.

F-2.0 APPLICABLE DOCUMENTS

The following documents of the issue in effect on the date of May 3, 1976 form a part of this specification to the extent specified herein. In the event of conflict with the documents referenced herein and the contents of this specification, the contents of this specification shall be considered a superseding requirement.

SPECIFICATIONS

Military

CDRL A011

For BFTSC Program, F04701-75-C-0149, Computer Program Development Specification BFTSC Executive/Recovery Software Design.

CDRL A007

Brassboard Fault Tolerant Spaceborne Computer (BFTSC) (F04701-75-C-0149) Configuration Item Development Specification BFTSC Architecture Design.

SAMSO SS-SDSF-01.0

Reports

ER75-4400

Users Programming Information for the Brassboard Fault Tolerant Computer.

System Specification for Interim Operational Configuration Digital Logic Emulation System

Nanodata, Programmable Run-Time Operater Display Users Guide

Nanodata, Task Control Program Overview

F-3.0 REQUIREMENTS

The FTSC instructions will be emulated to provide bit for bit agreement with the operation of the FTSC computer. Two memory modules and the active CPU will be simulated. The FTSC I/O operations will emulate the transfer of data and commands between the CPU and I/O ports. The simulation of the actual external I/O devices may be achieved through user-coded routines called by the emulator. The FTSC emulator will provide the necessary linkage to add user-coded device simulations. The interrupt logic of the FTSC will be emulated, although not all the interrupt conditions may be generated on the emulation. The FTSC execution time clock will be approximated by a count of the number of FTSC instructions executed. This simulated clock will be scaled at 120,000 instructions per second, which is the average instruction rate of the Brassboard Fault Tolerant Spaceborne Computer

F-3.1 Computer Program Definition

F-3.1.1 CPU Emulation

The FTSC emulation will provide a functional simulation of all the FTSC instructions except those which are fault related. Table 3.1.1-1 defines the FTSC instruction set to be implemented. The STH, LMO, SBAO, SBAZ, SBDC, SBDZ instructions are considered to be fault related and will be treated as no-op instructions. These instructions either induce fault effects, are used solely to recover from fault conditions, or are used only in the monitor mode. The remaining instructions noted in Table 3.1.1-1 will be

emulated to perform the operations described by References ER75-4400 and TBD.

The emulation will maintain all quantities which are apparent to the FTSC software, providing bit-for-bit agreement between emulated instructions with the operation of FTSC hardware (excluding time dependent or fault related sequences). Internal FTSC quantities which are not directly available to the FTSC software will not be maintained. These include the internal busses, working registers, data encoding, and timing/response lines. The FTSC emulation will maintain the following items:

2 modules of Main Memory

8 General Purpose Registers

Program Counter

Extension Register

Overflow Status

Carry out Status

Interrupt Enable/Disable Status

The normal operation of the active FTSC processor will be emulated. No hardware faults will be simulated.

The required operation of the emulation in response to certain anomalies which may be classified as software faults is described in Table 3.1.1-2. This table defines the emulator processing when conditions occur which do not have a well defined result.

F-3.1.2 Input/Output (I/O) Emulation.

The FTSC emulator shall provide a functional emulation of the same number of serial interface ports and direct memory access channels as the Brassboard Fault Tolerant Spaceborne Processor. Data shall be passed to or accepted from external, user supplied, peripheral device simulation routines through shared blocks of QM-system memory (I/O ports). Calls to these external user routines, which are referred to as User I/O simulation routines or external interface events, may be initiated by the TASK/PROD system controlling the emulator to facilitate user initiated external I/O interrupts and, periodically, by the emulator.

F-3.1.2.1 Serial Interface Subsystem.

The serial interface subsystem emulon shall transfer data directly between the simulated FTSC memory and the simulated serial I/O ports, without attempting to mimic the timing and sequencing of signals on the FTSC serial interface buss. Serial I/O ports will simulate the data transfer between the FTSC Device Interface Units and the peripheral devices.

F-3.1.2.2 Device Initiated Interrupts.

The emulator shall examine the interrupt request signal at each serial I/O port following each external interface event. The higher priority serial I/O interrupt or other processing, as applicable, will be initiated at this time.

F-3.1.2.3 Block Transfers.

Upon initiation by the emulator, block transfers of data between the simulated serial I/O ports and the simulated FTSC memory will proceed until an acknowledge signal is required from one of the ports. The emulator will then resume instruction emulation until the next external interface event. Following each external interface event the transfer of I/O data will proceed until another acknowledge signal is required. Simulated serial block transfers shall proceed in this intermittent manner until the end of the block, at which time the serial I/O end of block interrupt shall be initiated.

F-3.1.2.4 Direct Memory Access Module.

The FTSC emulator shall emulate a DMA port similar to those provided for serial I/O, with request - acknowledge communications carried out through a set of status flags similar in function to the control lines to the FTSC. The simulated DMA port will provide for the transfer of up to (TBD) words of information between the emulator and the external routines at each external interface event. Once a simulated DMA transfer has been initiated, up to (TBD) words will be transferred to each external interface event until the requested number of words have been processed, at which time a DMA end of block interrupt will be initiated.

F-3.1.2.5 I/O Status Words.

The I/O status words, soft addresses F440 through F445, shall be maintained as applicable to permit FTSC software to monitor the status of an I/O operation in progress.

F-3.1.3 Interrupts.

All ten FTSC interrupt levels shall be supported. The following table defines the initiating conditions for each interrupt.

Fault

TBD

Power Down/Illegal OP Code

Illegal OP Code detected during instruction decode.

Arithmetic Fault

As defined for disallowed FTSC arithmetic operations.

Real Time interrupt

Initiated each (TBD)th FTSC instruction.

SIU (General)

Initiated upon user request.

DMA1 (General)

Initiated upon user request.

DMA2 (General)

Not emulated.

SIU (End of block)

Serial I/O block transfer word count satisfied.

DMA1 (End of block)

DMA word count satisfied.

DMA2 (End of block)

Not emulated.

F-3.1.4 Deferred Emulation Items.

Any features described in the performance and interface description sections of reference 1 are beyond the scope of the initial FTSC emulation. The emulation shall be designed to add these capabilities at a later time. The following deferred items are discussed in the order they appear in Reference CDRL A007.

F-3.1.4.1 BFTSC System Limitations.

The FTSC emulation will emulate the active CPU and the two active memory modules. The Configuration Control Unit shall not be simulated because it is used only to control the system during fault conditions. The Reconfiguration Read Only Memory is used only for hardware recovery and so is beyond the scope of this effort. The Bus Arbiter is an internal computer mechanism which shall not be considered in the functional level emulation. The Power Module, Circumvention Unit, and Timing Module shall not be simulated. The internal FTSC buses shall be emulated only to the extent that they are needed to perform the functional emulation.

F-3.1.4.2 Internal Bus Configuration.

The error code appended to the data and address words shall not be simulated. The byte rippler and cyclic code generation logic are outside of the scope the functional emulation. The control lines between the CPU and the memory modules are to be emulated to the extent that an invalid address is recognized.

F-3.1.4.3 Fault Recovery and Reconfiguration.

Section 3.2.3 of CDRL A007, Fault Recovery and Reconfiguration, is beyond the scope of the initial FTSC emulation.

F-3.1.4.4 Functional Characteristics.

The FTSC emulation shall emulate the functional characteristics of the FTSC computer with certain limitations. The FTSC emulation by itself will generate the conditions to trigger only Illegal OP Code, Arithmetic, and Real Time interrupts. The user I/O

simulation routines may cause the emulator interface routine to generate an I/O interrupt request. The Real Time interrupt shall be generated after a specified number of FTSC instructions have been executed as an approximation to the actual timing. The FTSC working registers and control registers shall be simulated only to the extent that they are required for the functional emulation. Their values need not be kept unless an FTSC instruction uses that variable explicitly.

F-3.1.4.5 System Modules

Most of the items described in section 3.2.5 of CDRL A007, System Modules, are beyond the scope of the initial effort. The internal busses, special function decoders, working registers, and microprogram operation are beyond a functional level emulation. The CCU, CPU/Bus state sequencer, Hardware Status Word generator, and watchdog timer will not be simulated. The fault interrupt, fault category, reconfiguration control, CCU input, and CCU output shall not be simulated since they are fault related items.

F-3.1.4.6 System Software

The FTSC emulation is not required to execute fault recovery or fault testing FTSC software. The emulation is not required to execute FTSC software requiring an interface to an I/O device for which no appropriate user device simulation is linked to the emulator.

F-3.1.5 FTSC Emulation Interface

The initial FTSC emulation will interface directly or indirectly with several other software elements. As the emulation task on the QM-1, it must interface with the TASK/PROD operating system. The object code generated by the SMITE compiler must be loaded into the QM-1. The FTSC program to run on the emulator must be read from a file into the simulated memory region on the QM-1. The FTSC emulator must also communicate with the Interim Control System and user programs written to supply input/output device simulations. This interface is illustrated in Figure 3.5.1-1.

F-3.1.5.1 SMITE Interface

A CDC CYBER program will process the output of the SMITE compiler to generate a magnetic tape containing the

control store image of the emulator code and flags needed to load and run the emulator. A function will be added to the QM-1 operating system to read the SMITE tape and initiate the emulator.

F-3.1.5.2 FTSC Assembler/Loader Interface

A CDC CYBER program will be developed to process the output of the FTSC Assembler/Loader (ER75-4400) and create a magnetic tape which can be read into the QM-1. The format of this tape shall be made acceptable to the PROD function LOADMT. No modifications to the QM-1 operating system are required for the FTSC memory load interface.

F-3.1.5.3 QM-1 Operating System Interface

The QM-1 emulation system has three inter-related elements. The TASK system controls the scheduling, physical I/O operations, interrupt processing, and PROD/emulator communication. The PROD system interfaces the emulation with the operator, contains debugging features, display routines, simulation control commands, and the means of communicating with the I/O device simulation routines. The emulator must interface with the TASK and PROD systems, particularly regarding the location of interface and display variables and the protocol for transfering control.

The changes to TASK and the basic PROD system shall be minimized to avoid problems at the time of TASK/PROD system updates and the later transition to the SDSF site. Required operating system changes should be implemented by extensions to the PROD system. The following operator commands shall be added for the FTSC emulation:

PC

Set FTSC program counter

LSMITE

Load SMITE object code into QM-1 control store

DSMITE

Dump control store onto magnetic tape

REG

Modify an FTSC register

PORT

Modify an FTSC simulated port

STAT

Modify an I/O status line

WP

Set Write Protect Boundary in a memory module

An FTSC state display routine will be developed to output the following quantities to the extent that they are needed to debug the FTSC emulator:

FTSC Program Counter

FTSC Instruction Register

8 FTSC General Purpose Registers

FTSC Status as applicable

Instruction Count

Operand Address

Operand Value

The layout of the state display is shown in Figure 3.1.5-2.

A TASK/PROD routine shall be developed to extend the basic PROD capabilities for storing and displaying simulated memory, stepping the target program through a complete emulated instruction, and breakpointing an emulated instruction word for the FTSC emulation. The emulator and the QM-1 system shall be consistent about the location of interface variables and external calling sequences so that the basic PROD capabilities work. A control procedure shall be developed to assure that the SMITE code satisfies these interface requirements.

F-3.1.5.4 User Routine Interface

The user I/O simulation routine shall act as a PROD subroutine called by the emulator. The FTSC effort shall document PROD register conventions required for the user routine, PROD utility routines available to the user, conventions for using the PROD utility routines, and the emulator linkage conventions.

Appendix G: Recommended SMITE Coding Conventions and Standards

This appendix documents standards and conventions TRW has found useful in the development of SMITE computer descriptions and emulations. Two separate issues are presented: the development and maintenance of the emulator data base, and standards of coding style.

G.1 Data Base Development and Maintenance

Names chosen for identifiers in the computer description should conform to the naming established for the computer being described whenever possible. Where additional items are required, such as temporary registers not available to the target machine assembly language programmer, they should be named to be descriptive of their function.

Commentary should also be used to define the exact function of a variable when appropriate. Examples of the suggested commentary may be found in the FTSC description, Appendix H.

The size of the microcode may be reduced by declaring the most used (i.e. referred to by the most number of statements) variables first in the description so that the compiler allocates them to the first 64 words of QM-1 memory. This allows the compiler to use the short LDI instruction when generating address references, rather than the longer LDN instruction.

When a DEFAULT declaration is used, it should be the first data item declared in the main processor.

G.2 Coding Style

Each processor should be headed or prefaced by a sequence of descriptive and explanatory comments. These comments should address the topics found in MIL-STD-490-B5 (Paragraph 3) documentation; indeed, for the FTSC description, the relevant portions of the MIL-STD-490 specification was coded directly into the description. As a minimum, a well commented processor description should include the following:

1. The inputs to and outputs from the processor, taking particular notice of the type and format of the variables.

2. A description of the use made of any global data base variables referenced or set.

3. A functional description of the processing performed within the processor, including error conditions returned or fielded.

The number of comments and the degree of detail in the comments is a function of individual style and the complexity of the computer being described. For example, the Intel 8080 description (Appendix C) contains relatively few comments due to the simplicity of the machine and the resulting clarity of the description. The FTSC, in comparison, is a much more complex computer, and therefore the description benefited from the heavy use of comments.

A primary goal of SMITE coding style should be the clarity and readability of the computer description. Writability should be a secondary, if not irrelevant, consideration. Therefore, the description should utilize indenting to delineate levels of declarations and control flow, and should incorporate blank lines between logically distinct units of the description of suggest the distinction present.

Indentation in declaration statements should be used to indicate relationships created by he use of the DEFINED phrase. For example,

```
DECLARE
   REGS[0:3]<15:0>,
      REGISTERS[0:7]<7:0> DEFINED REGS,
         B DEFINED REGISTERS[0],
         C DEFINED REGISTERS[1],
         D DEFINED REGISTERS[2],
         E DEFINED REGISTERS[3],
         H DEFINED REGISTERS[4],
         L DEFINED REGISTERS[5],
         STATUS DEFINED REGISTERS[6],
            CARRY FLAG DEFINED STATUS<0>,
            PARITY FLAG DEFINED STATUS<2>,
            AUX-CARRY DEFINED STATUS<4>,
            ZERO FLAG DEFINED STATUS<6>,
            SIGN DEFINED STATUS<7>,
         A DEFINED REGISTERS[7],
            CARRY-A DEFINED REGS[3]<8:0>;
```

describes the main register set of the Intel 8080. The use of indentation clearly shows the use of subfields and the dependency of subfield definitions on their parents. The job of tracing a series of subfield definitions to the ultimate top-level parent is also simplified by this visual technique.

Subprocessor structure is also usefully shown by he selective use of indentation. For example, the (partial) structure of the Intel 8080 description is

```
INTEL-8080: PROCESSOR;
   ADD-REG: PROCESSOR;
      ADD-REG: END;
   GETADD: PROCESSOR;
      GETADD: END;
   .
   .
   .
   INTEL-8080: END;
```

The above example also shows the suggested way for using indentation for PROCESSOR/END, BEGIN/END, CASE/END CASE, etc. Thus,

```
GETADD: PROCESSOR;

   GETADD: END;
```

for processors,

```
BEGIN;

   END;
```

for BEGIN/END,

```
CASE OPCODE;

   END CASE;
```

for case statements, and similarly for the others. IF statements can be handled in this way as well:

```
IF INTERRUPT AND INTE
   THEN BEGIN;
      INTERRUPT <- INTE <- 0;
      IR <- INTERRUPT-IR;
      END;
   ELSE BEGIN;
```

```
        IR <- MEM[PC];
        PC <- PC + 1;
        END;
     END IF;
```

Context labels can be integrated easily into this scheme:

```
   CODE10: BEGIN;
      OP-REG <- OPERAND;
      IN 3 PERFORM-OP;
      CODE10: END;
```

or

```
   JMP: IN 10 BEGIN;
      GETADD;
      PC <- OP-PAIR;
      END;
```

Finally, IN statements may be treated as prefixes to other statements, with no effect on indenting, as illustrated in the last example.

Appendix H: FTSC Description

```
''        FFFFFFFF  TTTTTTTT   SSSSSSS   CCCCCC                    ''
''        FF           TT     SS        CC    CC                   ''
''        FF           TT     SS        CC                         ''
''        FFFF         TT      SSSSSS   CC                         ''
''        FF           TT           SS  CC                         ''
''        FF           TT           SS  CC    CC                   ''
''        FF           TT     SSSSSSS    CCCCCC                    ''

''        FAULT-TOLERANT SPACEBORNE COMPUTER                       ''

FTSC:PROCESSOR;
     DECLARE
     DEFAULT<31:0>  REGISTER,              '' NOMINAL ATTRIBUTES    ''
    PAR1<35:0>,                            '' MERGED FLAG WORD 1    ''
     STEP-FLAG FLAG DEFINED PAR1<35>,      '' INTERFACE WITH STEP COMMAND ''
     MPY-SIGN  FLAG DEFINED PAR1<32>,      '' SIGN OF MULTIPLY RESULT ''
     NORM-SIGN FLAG DEFINED PAR1<31>,      '' SIGN OF VALUE TO NORMALIZE ''
     AMZERO    FLAG DEFINED PAR1<30>,      '' ADDRESS MODE ZERO FLAG ''
     SHIFT-DIRECTION FLAG DEFINED PAR1<29>,
     EX-SIGN FLAG DEFINED PAR1<28>,        '' SIGN OF EXTENSION REGISTER ''
                                           '' SET TO 1 FOR LEFT SHIFTS ''
     EXECUTE FLAG DEFINED PAR1<10>,        '' EXECUTE INSTRUCTION FLAG ''
''        INTERRUPT STUFF                                          ''

     INTREQ<9:0>    DEFINED PAR1<9:0>, '' INTERRUPT REQUEST FLAGS  ''
      FAULT-REQ FLAG DEFINED   PAR1<9>,'' FAULT INTERRUPT REQUEST  ''
      POWER-REQ FLAG DEFINED   PAR1<8>,'' POWER DOWN INTERRUPT REQUEST''
      ARITH-REQ FLAG DEFINED   PAR1<7>,'' ARITHMETIC ERROR INT REQUEST''
      RTIME-REQ FLAG DEFINED   PAR1<6>,'' REAL TIME INTERRUPT REQUEST ''
      SIUG-REQ  FLAG DEFINED   PAR1<5>,'' SIU GENERAL INTERRUPT REQ ''
      DMA1G-REQ FLAG DEFINED   PAR1<4>,'' DMA1 GENERAL INT REQUEST  ''
      DMA2G-REQ FLAG DEFINED   PAR1<3>,'' DMA2 GENERAL INT REQUEST  ''
      SIUE-REQ  FLAG DEFINED   PAR1<2>,'' SIU END OF BLOCK INTERRUPT ''
      DMA1E-REQ FLAG DEFINED   PAR1<1>,'' DMA1 END OF BLOCK INT REQ ''
      DMA2E-REQ FLAG DEFINED   PAR1<0>,'' DMA2 END OF BLOCK INT REW ''


    PAR2,                                  '' SECOND MERGED DATA WORD ''
    STATUS<7:0> DEFINED PAR2<25:18>,       '' STATUS REGISTER       ''
      ENI-STAT FLAG DEFINED STATUS<7>,     '' ENABLE INTERRUPT NETWORK ''
      DIVCK    FLAG DEFINED STATUS<6>,     '' DIVIDE CHECK STATUS   ''
      OVERFLOW FLAG DEFINED STATUS<5>,     '' OVERFLOW STATUS       ''
```

```
    ILLOP    FLAG DEFINED STATUS<4>, '' ILLEGAL OPCODE STATUS      ''
    CARRY    FLAG DEFINED STATUS<3>, '' CARRYOUT STATUS            ''
    INTLEV<2:0>   DEFINED STATUS<2:0>,'' INTERRUPT LEVEL           ''
  PC<15:0>     DEFINED PAR2<15: 0>,   '' PROGRAM COUNTER           ''


''        FTSC REGISTERS                                           ''

  GPXR [0:7],                         '' GENERAL PURPOSE REGISTER S ''
  EX36<35:0>,                         '' EXTENSION REGISTER 36 BITS ''
   EX DEFINED EX36<31:0>,             '' FTSC EXTENSION REGISTER    ''
   OVF-EX<1:0> DEFINED EX36<33:32>,   '' 2 OVERFLOW BITS - FTSC MPY ''
  INR,                                '' INSTRUCTION REGISTER       ''
    OPCODE<6:0>   DEFINED INR<31:25>,'' OPERATION CODE              ''
    RB<2:0>       DEFINED INR<24:22>,'' OPERAND RB REGISTER         ''
    RA<2:0>       DEFINED INR<21:19>,'' OPERAND RA REGISTER         ''
    AM<2:0>       DEFINED INR<18:16>,'' ADDRESSING MODE             ''
    ADDRESS<15:0> DEFINED INR<15:0>, '' ADDRESS OR IMMEDIATE DATA   ''

  PAR3,                               '' THIRD MERGED PARENT WORD   ''
   INPROC <9:0> DEFINED PAR3<27:18>, '' INTERRUPT IN PROCESS FLAGS ''
   INTMASK<9:0> DEFINED PAR3< 9: 0>, '' INTERRUPT MASK             ''


''        WORKING STORAGE                                          ''

  OPERAND33<32:0>,                    '' 33 BIT INSTRUCTION OPERAND ''
       OPERAND<31:0> DEFINED OPERAND33<31:0>,
                                      '' INSTRUCTION OPERAND        ''
  EFFAD32,                            '' 32 BIT ADDRESS (INDIRECTS) ''
   EFFAD<17:0> DEFINED EFFAD32<17:0>,'' OPERAND ADDRESS            ''

  REG-OP33<32:0>,                     '' GPXR(RB) OPERAND,RESULT    ''
    REG-OP<31:0> DEFINED REG-OP33<31:0>,

  WORK1,                              '' TEMPORARY 32 BIT CELL      ''
  WORK2,                              '' TEMPORARY 32 BIT CELL      ''
  WORK3,                              '' TEMPORARY 32 BIT CELL      ''
  ICOUNT,                             '' FTSC INSTRUCTION COUNT     ''
  SHIFT-MASK33<32:0>,                 '' N BIT MASK, N=SHIFT COUNT  ''
   SHIFT-MASK<31:0> DEFINED SHIFT-MASK33<31:0>,
```

```
    P4<35:0>,                                  '' 4TH PARENT QM1 WORD          ''
    SHIFT-COUNT<17:0> DEFINED P4<35:18>,'' COUNTER FOR DBL SHIFT PASS ''
    PASS-COUNT<17:0>  DEFINED P4<17:0>,'' DBL SHIFT COUNTER (LE 32)   ''

    P5<35:0>,                                  '' 5TH PARENT QM1 WORD          ''
    OPERAND-EXP<17:0> DEFINED P5<35:18>,'' EXPONENT OF REG-OP          ''
    REG-OP-EXP<17:0>   DEFINED P5<17:0>,'' EXPONENT OF OPERAND         ''

    P6<35:0>,                                  '' 6TH PARENT QM1 WORD          ''
    NORMALIZE-COUNT<17:0> DEFINED P6<35:18>,
                                               '' NUMBER OF SHIFTS TO NORMALIZ''

    P7<35:0>,
     TYPE<17:0> DEFINED P7<35:18>,             '' WRITE PROTECT THIS MODULE    ''
     MEMTYPE<17:0> DEFINED P7<17:0>,           '' BIT CODED WRITE PROTECT WORD''

''        I/O STUFF                                                           ''

    IO-STATUS[0:5],                            '' I/O STATUS WORDS             ''
     DMA1-STAT1  DEFINED IO-STATUS[0],
     DMA1-STAT2  DEFINED IO-STATUS[1],
     DMA2-STAT1  DEFINED IO-STATUS[2],
     DMA2-STAT2  DEFINED IO-STATUS[3],
     SIU-STAT1   DEFINED IO-STATUS[4],
     SIU-STAT2   DEFINED IO-STATUS[5],

''        MEMORY STUFF                                                        ''

    MEM[0:12287] MEMORY;                       '' 3 MEMORY MODULES             ''



''        SMITE DATA STRUCTURES    ''
DECLARE
    WORD<31: 0> DATA,
        MANTISSA<23: 0> DATA DEFINED WORD<31:8>,'' F.P. MANTISSA      ''
        EXPONENT< 7: 0> DATA DEFINED WORD< 7: 0>,'' F.P. EXPONENT     ''
        INDEX<17:0> DATA DEFINED WORD<17:0>,     '' REGISTER INDEX VAL''
```

```
        EXP-SIGN          DATA DEFINED WORD<7>,    '' EXPONENT SIGN BIT ''
        EXP-CARRY         DATA DEFINED WORD<8>,    '' EXPONENT CARRY    ''

        SIGN-BIT          DATA DEFINED WORD<31>,   '' FTSC SIGN BIT     ''
        BIT-30            DATA DEFINED WORD<30>,   '' FTSC BIT 30       ''
        BITS-31-30<1:0>   DATA DEFINED WORD<31:30>,'' FTSC BITS 30,29   ''
        BITS-30-29<1:0>   DATA DEFINED WORD<30:29>,'' FTSC BITS 30,29   ''
        LOW-BIT           DATA DEFINED WORD<0>,    '' FTSC LOW ORDER BIT''
        BIT1              DATA DEFINED WORD<1>;

''        EXTERNAL PROCESSORS       ''
DECLARE  OP-STEP EXTERNAL,                ''FTSC INSTRUCTION STEP''
         OP-HALT EXTERNAL,                ''HALT INSTRUCTION ''
         OP-ERROR EXTERNAL,               ''INSTRUCTION ERROR''
         OP-NOTSIM EXTERNAL,              ''UNSIMULATED INSTRUCTION ''
         IN-PORT PORT,                    ''CRT INPUT ''
         OUT-PORT PORT,                   '' CRT OUTPUT ''
         PUTBUF EXTERNAL ;
DECLARE IO-CONTROL EXTERNAL  ; '' TEMPORARY FOR NOW ''


''*      3.2.2     OPERAND-FETCH
***
***               OPERAND-FETCH FORMS THE INSTRUCTION OPERAND AND
***                                THE OPERAND ADDRESS.
***
***      3.2.2.1   INPUTS -- ADDRESS CONTAINS THE 16 BIT ADDRESS FIELD
***                                OF THE INSTRUCTION.
***                           AM      CONTAINS THE ADDRESSING MODE.
***                           RA      POINTS TO THE INDEX REGISTER.
***
***      3.2.2.2   PROCESS-- THE EFFECTIVE ADDRESS IS SET TO THE ADDRESS
***                             FIELD OF THE INSTRUCTION.
***
***                           A BRANCH BASED ON THE ADDRESSING MODE IS TAKEN.
***
***                           0 - REGISTER TO REGISTER
***                                OPERAND = CONTENTS OF RA REGISTER
***                                SET AMZERO TRUE.
```

```
***
***                              1 - IMMEDIATE
***                                  OPERAND = SIGN EXTENDED ADDRESS FIELD.
***
***                              2 - DIRECT
***                                  OPERAND = CONTENT OF EFFECTIVE ADDRESS
***
***                              3 - INDIRECT
***                                  EFFECTIVE ADDRESS = MEM(ADDRESS)
***
***                              4 - INDEXED,POST INCREMENT
***                                  EFFECTIVE ADDRESS = RA REGISTER + ADDRESS
***                                  RA REGISTER IS INCREMENTED BY 1.
***
***                              5 - INDEXED,PRE  DECREMENT
***                                  RA REGISTER IS DECREMENTED BY 1.
***                                  EFFECTIVE ADDRESS = RA REGISTER + ADDRESS
***
***                              6 - INDEXED
***                                  EFFECTIVE ADDRESS = RA REGISTER + ADDRESS
***
***                              7 - INDEXED INDIRECT
***                                  EFFECTIVE ADDRESS = MEM(RA + ADDRESS)
***
***                             FOR MODES 2-7, OPERAND = CONTENTS OF EFFECTIVE ADDR
***
***       3.2.2.3   OUTPUTS-- EFFAD IS THE EFFECTIVE ADDRESS
***                           OPERAND IS THE INSTRUCTION OPERAND.
***                           AMZERO IS SET TRUE IF THE REGISTER-REGISTER MODE
***                           REGISTER MODIFIED IN MODE 4 OR 5  ''
OPERAND-FETCH:     PROCESSOR;
''         EFFECTIVE ADDRESS COMPUTATION                                    ''

     EFFAD <- ADDRESS;
    CASE AM;
         BEGIN;                          '' MODE 0 REGISTER-REGISTER    ''
             OPERAND <- GPXR[RA] ;
             AMZERO <- 1;
         END;

         OPERAND <- SE(ADDRESS);         '' MODE 1 IMMEDIATE            ''
```

```
            OPERAND <- FETCH(EFFAD);          '' MODE 2 DIRECT ADDRESSING   ''

            BEGIN;                            '' MODE 3 INDIRECT            ''
                EFFAD32 <- FETCH(EFFAD) ;
                OPERAND <- FETCH(EFFAD);
            END;

           BEGIN;                             '' MODE 4 INDEXED, POST INCR  ''
                EFFAD <- EFFAD + GPXR[RA].INDEX ;
                GPXR[RA] <- GPXR[RA] + 1;
                OPERAND <- FETCH(EFFAD) ;
            END;

            BEGIN;                            '' MODE 5 INDEXED, PRE DEC    ''
                GPXR[RA] <- GPXR[RA] - 1;
                EFFAD <- EFFAD + GPXR[RA].INDEX ;
                OPERAND <- FETCH(EFFAD);
            END;

            BEGIN;                            '' MODE 6 INDEXED             ''
                EFFAD <- EFFAD + GPXR[RA].INDEX ;
                OPERAND <- FETCH(EFFAD) ;
            END;

            BEGIN;                            '' MODE 7 INDEXED INDIRECT    ''
                EFFAD <- EFFAD + GPXR[RA].INDEX ;
                EFFAD32 <- FETCH(EFFAD) ;
                OPERAND <- FETCH(EFFAD);
            END;

        END CASE;

    OPERAND-FETCH:     END;



    ''        PERFORM MEMORY FETCH REFERENCES                              ''

    ''*     3.2.3.1   FETCH PROCESSOR(ADDRESS)
    ***
    ***               FETCH READS DATA FROM EMULATED MEMORY TO AN ASSIGNED VARIABLE
```

```
***
***       3.2.3.1.1 INPUTS -- THE CALLING PARAMETER TO FETCH IS THE
***                           FTSC MEMORY ADDRESS.
***
***       3.2.3.1.2 PROCESS-- IF THE ADDRESS IS BETWEEN 0 AND 8191,
***                               THE DATA IS FETCHED FROM MEMORY.
***                           IF THE ADDRESS IS NON-EXISTENT A VALUE OF
***                               ZERO IS RETURNED.
***                           IF THE ADDRESS IS IN MODULE F, THE SIMULATED
***                               VALUE IS RETURNED (RESIDES AFTER
***                               SIMULATED MODULE 1).
***                           IF THE ADDRESS IS AN I/O STATUS LOCATION,
***                               STATUS WORD IS RETURNED
***
***       3.2.3.1.3 OUTPUTS-- THE SIMULATED VALUE AT THE FTSC ADDRESS IS
***                               LOADED INTO THE ASSIGNMENT VARIABLE.
***                           I.E. THE USAGE OF FETCH IS AS FOLOWS
***                               VARIABLE <- FETCH(ADDRESS);
***                               THIS RESEMBLES A FORTRAN FUNCTION    ''
FETCH:PROCESSOR<31:0>(ADDR);
    DECLARE ADDR<17:0> ;
    IF ADDR<15:0> < X'2000'
         THEN FETCH <- MEM[ ADDR<15:0> ] ;
         ELSE BEGIN;
              IF ADDR<15:0> < X'F000'
                   THEN BEGIN;
                        FETCH <- 0;
                        END;
                   ELSE BEGIN;
                        FETCH <- MEM[ ADDR<11:0> + X'2000' ] ;
                        IF(ADDR<15:0> >= X'F440') AND
                          (ADDR<15:0> <= X'F445')
                             THEN FETCH <- IO-STATUS[ ADDR<2:0> ] ;
                        END IF;
                        END;
                   END IF;
              END;
         END IF;
FETCH:END;
```

```
''*       3.2.3.2   FETCH-MULTIPLE (NUMBER)
***
***                 FETCH-MULTIPLE PERFORMS A VECTOR LOAD INTO THE
***                                FTSC REGISTER SET
***
***       3.2.3.2.1 INPUTS -- THE CALLING PARAMETER INDICATES THE
***                           LENGTH OF THE LOAD VECTOR
***
***                           EFFAD IS THE EFFECTIVE OPERAND ADDRESS
***                           AMZERO IS NON-ZERO FOR REGISTER-REGISTER MODE
***
***       3.2.3.2.2 PROCESS-- THE RESULT OF THE OPERAND FETCH PROCESSOR
***                               IS MOVED TO GPXR(RB).
***                           A LOOP IS PERFORMED TO MOVE THE REMAINING
***                               ELEMENTS (NUMBER-1).
***                           IN THIS LOOP THE PROCESSOR LOAD-NEXT IS CALLED
***                               TO FETCH THE NEXT MEMORY/REGISTER VALUE
***                               INTO REG-OP.  THIS VALUE IS THEN MOVED
***                               TO GPXR(RB).
***
***       3.2.3.2.3 OUTPUTS-- GPXR CONTAINS THE RESULT OF THE VECTOR LOAD
***                           REG-OP CONTAINS THE LAST VALUE LOADED.
                              RA IS INCREMENT BY NUMBER-1
                              RB IS INCREMENT BY NUMBER-1
                              EFFAD IS INCREMENTED BY NUMBER-1          ''
FETCH-MULTIPLE:     PROCESSOR(N);
    DECLARE N<17:0> ;
    GPXR[RB] <- OPERAND;
    DO FOR 2 TO N;                                  '' N-1 PASSES       ''
       GPXR[RB] <- REG-OP <- LOAD-NEXT;
FETCH-MULTIPLE:     END;



''*       3.2.3.3   LOAD-NEXT
***
***                 LOAD-NEXT LOADS THE NEXT OPERAND VECTOR FROM
***                      THE REGISTER FILE OR MEMORY
***
***       3.2.3.3.1 INPUTS -- EFFAD IS THE CURRENT EFFECTIVE ADDRESS
```

```
***                          RA    IS THE CURRENT OPERAND REGISTER
***                          AMZERO IS THE REGISTER-REGISTER MODE FLAG
***
***      3.2.3.3.2 PROCESS-- BUMP RA,RB, AND EFFAD BY 1
***                          IF AMZERO IS SET, LOAD GPXR(RA)
***                               OTHERWISE LOAD MEM(EFFAD)
***
***      3.2.3.3.3 OUTPUTS-- RA,RB,EFFAD ARE INCREMENTED BY 1
***                          THE REGISTER/MEMORY OPERAND IS LOADED
***                               INTO THE ASSIGNED VARIABLE.
***                               I.G. VARIABLE C- LOAD-NEXT;          ''
'''                          REG-OP IS LOADED WITH NEXT REGISTER VALUE''
LOAD-NEXT:          PROCESSOR<31:0> ;
    REG-OP <- GPXR[RB <- RB +1 ] ;
    IF AMZERO
         THEN BEGIN;
              RA <- RA+1;
              LOAD-NEXT <- GPXR[RA];
              END;
         ELSE BEGIN;
              EFFAD <- EFFAD + 1;
              LOAD-NEXT <- FETCH(EFFAD);
              END;
         END IF;
LOAD-NEXT:          END;


'''      3.2.3.4   LDN
***
***               LDN LOADS THE TWOS COMPLEMENT OF THE INSTRUCTION
***               OPERAND INTO THE RESULT REGISTER AND TEST FOR OVERFLOW.
***
***      3.2.3.4.1 INPUTS -- OPERAND CONTAINS THE INSTRUCTION OPERAND.
***
***      3.2.3.4.2 PROCESS-- THE NEGATIVE OF OPERAND IS STORED IN REG-OP.
***                          IF THE VALUE HAS ONLY THE SIGN BIT SET,
***                             THEN SET-OVERFLOW IS CALLED.
***
***      3.2.3.4.3 OUTPUTS-- REG-OP CONTAINS THE RESULT.
***                          OVERFLOW IS SET IF THE OPERAND = 80000000''
LDN:                PROCESSOR;
```

```
    REG-OP <- - OPERAND ;
    IF REG-OP = X'80000000'
        THEN SET-OVERFLOW;
        END IF;
LDN:                END;


''*     3.2.3.5   LDNF
***
***               LDNF LOADS THE NEGATIVE OF THE INSTRUCTION OPERAND
***                    INTO THE RESULT REGISTER. IT ADJUSTS FOR NEGATIVE
***                    FLOATING POINT POWERS OF 2.0 AND TEST FOR OVERFLOW.
***
***     3.2.3.5.1 INPUTS -- OPERAND CONTAINS THE INSTRUCTION OPERAND.
***
***     3.2.3.5.2 PROCESS-- IF THE MANTISSA OF THE OPERAND IS 800000,
***                            THEN THE OPERAND EXPONENT IS INCREMENTED
***                            BY 1 AND THE MANTISSA IS SET TO 400000
***                         OTHERWISE, THE MANTISSA IS NEGATED AND STORED
***                            IN THE MANTISSA OF REG-OP.  THE EXPONENT
***                            OF THE OPERAND IS STORED IN THE EXPONENT
***                            OF THE RESULT REGISTER. ''
LDNF:               PROCESSOR;
    IF OPERAND.MANTISSA = X'800000'
        THEN BEGIN;
             REG-OP-EXP <- 1 + SE(OPERAND.EXPONENT) ;
             REG-OP <- SRL (OPERAND,1);
             TEST-FP-OVERFLOW;
             END;

        ELSE BEGIN;
             REG-OP <- OPERAND;
             REG-OP.MANTISSA <- - OPERAND.MANTISSA;
             END;
        END IF;
LDNF:               END;


''        MEMORY STORE PROCESSOR                                    ''
```

```
''*       3.2.3.6   STORE (VALUE,ADDRESS)
***
***                 STORE PLACES THE PARAMETER VALUE IN THE SIMULATED
***                       MEMORY ADDRESS INDICATED BY THE PARAMETER ADDRESS
***
***       3.2.3.6.1 INPUTS -- THE CALLING PARAMETERS INDICATE THE
***                             VALUE AND THE MEMORY ADDRESS.
***
***       3.2.3.6.2 PROCESS-- IF THE ADDRESS IS IN MODULE F :
***                             TYPE (WRITE-PROTECT) IS FORMED FROM
***                                  ONE OF THE 4 UPPER BITS OF MEMTYPE
***                             IF THE ADDRESS FALLS BETWEEN F440 AND
***                                  F445, THE I/O CONTROL PROCESSOR IS CALLED
***                             IF THE ADDRESS FALLS BETWEEN F404 AND F407,
***                                  INTMASK OR ENI-STAT ARE ALTERED.
***                             THE ADDRESS IS MAPPED TO A QM-1 ADDRESS
***
***                           IF THE ADDRESS DOES NOT EXIST, TYPE = 0
***                             NO ERROR MESSAGE AT THIS TIME.
***
***                           OTHERWISE, TYPE IS FORMED FROM MEMTYPE TO
***                             DESIGNATE THE WRITE PROTECT STATUS OF
***                             THAT QUARTER OF A MEMORY MODULE
***
***                           THEN ALL PATHS REJOIN
***                             IF TYPE IS NON-ZERO, THE INPUT VALUE
***                             IS STORED AT THE QM-1 ADDRESS CORRESPONDING
***                             TO THE INPUT ADDRESS.
***
***       3.2.3.6.3 OUTPUTS-- THE SIMULATED MEMORY ADDRESS IS CHANGED
***                             UNLESS THE WRITE PROTECT STATUS IS SET.
***                           INTMASK AND ENI-STAT ARE MODIFIED BY STORES
***                             TO F404-F407                              ''
STORE:PROCESSOR(VALUE,ADDR);
     DECLARE
     VALUE<31:0>,                        '' FTSC MEMORY DATA           ''
     ADDR<17:0>;                         '' FTSC MEMORY ADDRESS        ''

    WORK1.INDEX <- ADDR;
    IF ADDR<15:12> = X'F'
         THEN BEGIN ;
              TYPE <- SLL(1,ADDR<14:11> ) AND MEMTYPE ;
```

```
                WORK1<15:12> <- 2;
''      ASSUMES F4XX IS NOT WRITE PROTECTED ''
                IF ( ADDR<15:0> >= X'F440')
                AND (ADDR<15:0> <= X'F445')
                     THEN BEGIN;
                          '' SET UP VALUE AND ADDR AS PARAMETERS ''
                     IO-CONTROL;
                     END;
                     END IF;
                IF SRL( ADDR<15:0>-X'F404', 2) = 0
                     THEN BEGIN;
                          CASE ADDR<1:0> ;
                               INTMASK<7:0> <- VALUE<7:0> ;
                               NULL;
                               ENI-STAT <- 0 ;
                               ENI-STAT <- 1 ;
                          END CASE;
                     END;
                END IF;
                END;
           ELSE BEGIN;
                IF ADDR<15:12> > 1
                     THEN BEGIN ;
                          TYPE <- 0;
                          END;
                     ELSE TYPE <- SLL(1,ADDR<13:11>) AND MEMTYPE ;
                     END IF;
                END;
           END IF;
      IF TYPE /= 0
           THEN MEM[ WORK1<15:0> ] <- VALUE ;
       END IF;
STORE:END;
```



```
''*       3.2.3.7   STORE-DOUBLE
***
***                 STORE DOUBLE STORES A VALUE IN TWO ADJOINING MEMORY MODULES
***
***       3.2.3.7.1 INPUTS -- OPERAND CONTAINS THE DATA TO STORE.
```

```
***                              EFFAD CONTAINS THE OPERAND (STORE) ADDRESS
***                              AMZERO IS 1 FOR THE REGISTER-REGISTER MODE
***
***       3.2.3.7.2 PROCESS-- THE PROCESSOR IS BYPASSED IN THE REGISTER TO
***                               REGISTER MODE OF ADDRESSING.
***                              THE STORE PROCESSOR IS CALLED TWICE
***                                   OPERAND IS THE VALUE PARAMETER
***                                   EFFAD,EFFAD+4096 ARE THE ADDRESS PARAMETERS
***
***       3.2.3.7.3 OUTPUTS-- MEM(EFFAD), MEM(EFFAD + 4096) ARE STORED
***                               INTO BY THE STORE PROCESSOR          ''
STORE-DOUBLE:       PROCESSOR;
    IF NOT AMZERO
         THEN BEGIN;
              STORE(OPERAND,EFFAD) ;
              STORE(OPERAND,EFFAD + 4096);
              END;
         END IF;
STORE-DOUBLE:       END;



''*       3.2.3.8   STORE-MULTIPLE (NUMBER)
***
***                 STORE-MULTIPLE STORES A REGISTER VECTOR INTO TWO
***                               ADJOINING MEMORY MODULES.
***
***       3.2.3.8.1 INPUTS -- THE INPUT PARAMETER NUMBER IS THE LENGTH
***                               OF THE REGISTER VECTOR.
***                              EFFAD IS THE STORE ADDRESS.
***                              REG-OP INITIALLY EQUALS GPXR(RB).
***       3.2.3.8.2 PROCESS-- OPERAND AND REG-OP ARE LOADED WITH GPXR(RB).
***                              STORE-DOUBLE IS CALLED.
***                              RB AND EFFAD ARE INCREMENTED BY 1.
***                              THIS LOOP IS PERFORMED THE NUMBER OF TIMES
***                                   INDICATED BY THE CALLING PARAMETER.
***
***       3.2.3.8.3 OUTPUTS-- RB IS INCREMENTED BY NUMBER-1.
***                              EFFAD IS INCREMENT BY NUMBER-1.
***                              REG-OP = GPXR( LAST RB )
***
```

```
***                              THE MEMORY VECTORS BEGINNING AT MEM( OLD EFFAD)
***                                   AND MEM( OLD EFFAD +4096) ARE ALTERED
***                                                                        ''
STORE-MULTIPLE:     PROCESSOR(N);
    DECLARE N<17:0> ;
    OPERAND <- REG-OP;
    DO FOR 1 TO N;
         BEGIN;
              STORE-DOUBLE;
              EFFAD <- EFFAD + 1;
              RB    <- RB + 1;
              OPERAND <- REG-OP <- GPXR[RB];
         END;
STORE-MULTIPLE:     END;


''*       3.2.3.9    BAD-STORE
***
***                  BAD-STORE EMULATES THE SBAO,SBAZ,SBDO,SBDZ INSTRUCTIONS
***                       WHICH CAUSE MEMORY FAULT ERRORS TO OCCUR.
***
***       3.2.3.9.1 INPUTS -- AMZERO IS TRUE ONLY FOR THE REGISTER-REGISTER MODE.
***                           REG-OP CONTAINS THE CONTENTS OF REGISTER RB
***
***       3.2.3.9.2 PROCESS-- THE DATA IS STORED IF THE REGISTER TO REGISTER
***                               ADDRESSING MODE IS USED.
***                           OTHERWISE, THE FAULT REQUEST IS SET.
***       3.2.3.9.3 OUTPUTS-- THE REGISTER FILE IS MODIFIED IN THE REGISTER
***                               TO REGISTER MODE.
***                           OTHERWISE, NO STORE OCCURS.  ''
BAD-STORE:          PROCESSOR;
    IF AMZERO
         THEN GPXR[RA] <- REG-OP;
         ELSE FAULT-REQ <- 1;
         END IF;
BAD-STORE:          END;


ADD: PROCESSOR(ADDEND, ACO-FLAG);
     ''REPLACE NEXT STATEMENT AFTER COMPILER BUG IS FIXED.''
```

```
        ''DECLARE ACO-FLAG FLAG;''

        REG-OP33 <- (ACO-FLAG AND CARRY) +
             SE(ADDEND) + (REG-OP33 <- SE(REG-OP));
        IF (CARRY <- REG-OP33<32>) /= REG-OP.SIGN-BIT THEN
             SET-OVERFLOW;
        END IF;
ADD:    END;


''*       3.2.4.2   INTEGER MULTIPLY  PROCESSOR                    ''
''*
***                 MULTIPLY PERFORMS A 32 BIT BY 32 BIT MULTIPLICATION
***
***       3.2.4.2.1 INPUTS -- REG-OP  CONTAINS ONE MULTIPLIER
***                           OPERAND CONTAINS THE OTHER.
***                           NORMALLY THESE REPRESENT THE INSTRUCTION
***                             OPERAND AND AN FTSC REGISTER.
***
***       3.2.4.2.2 PROCESS-- THE ABSOLUTE VALUES OF THE MULTIPLIERS
***                             ARE FORMED.  A 2 BIT AT A TIME
***                             MULTIPLICATION IS PERFORMED TO PRODUCE
***                             THE 64 BIT PRODUCT.
***                           IF NECESSARY, THE PRODUCT IS THEN NEGATED.

***       3.2.4.2.3 OUTPUTS-- THE MOST SIGNIFICANT 32 BITS OF THE RESULT
***                             IS STORED IN REG-OP.  THE LEAST SIGNIFICANT
***                             32 BITS IS STORED IN EX.
***                           MPY-SIGN DENOTES THE SIGN OF THE RESULT.
***                           OPERAND IS LEFT UNCHANGED.               ''
MULTIPLY: PROCESSOR;
    EX <- 0 ;
    MPY-SIGN <- 0;
    IF OPERAND.SIGN-BIT
       THEN BEGIN;
            OPERAND <- - OPERAND ;
            MPY-SIGN <- 1;
            END;
       END IF;
    IF REG-OP.SIGN-BIT
```

```
        THEN BEGIN;
             REG-OP <- -REG-OP;
             MPY-SIGN <- NOT  MPY-SIGN;
             END;
        END IF;
    DO FOR 1 TO 16                          ;
        BEGIN ;
             EX36 <- SLL (EX36,2);
             CASE REG-OP <30:29> ;
                  NULL;
                  EX36 <- EX36 + OPERAND;
                  EX36 <- EX36 + SLL( OPERAND33,1 );
                  EX36 <- EX36 + SLL( OPERAND33,1 ) + OPERAND ;
             END CASE;
             REG-OP <- SLL (REG-OP,2 ) + OVF-EX ;
        END ;
    IF MPY-SIGN
        THEN BEGIN;
             EX <- - EX;
             IF EX = 0
                  THEN REG-OP <- - REG-OP;
                  ELSE REG-OP <- NOT  REG-OP;
                  END IF;
             END;
        END IF;
MULTIPLY:    END;


!!*      3.2.4.4   SHIFT PREPARATION PROCESSOR                          !!
!!*
***                SHIFT-PREP EXTRACTS THE SHIFT COUNT AND
***                DIRECTION FROM THE INSTRUCTION OPERAND.
***
***      3.2.4.4.1 INPUTS -- OPERAND CONTAINS THE MAGNITUDE AND
***                           DIRECTION OF THE SHIFT
***      3.2.4.4.2 PROCESS-- THE SIGN OF THE LOWER 8 BITS OF OPERAND
***                           IS TESTED. A NEGATIVE VALUE INDICATES A
***                           LEFT SHIFT.  THE MAGNITUDE OF THE LOWER
***                           8 BITS FORMS THE SHIFT COUNT.  THE SHIFT
***                           COUNT IS LIMITED TO 64 BITS
***      3.2.4.4.3 OUTPUTS-- SHIFT-DIRECTION IS 1 FOR LEFT SHIFTS
```

```
***                          SHIFT-COUNT IS THE SHIFT LENGTH (LIMITED
***                                   TO 64)                          ''

SHIFT-PREP:   PROCESSOR;
    SHIFT-COUNT <- SE(OPERAND.EXPONENT);
    IF SHIFT-COUNT <17>                              '' TEST NEGATIVE LEFT''
    THEN BEGIN;
         SHIFT-DIRECTION <- 1;
         SHIFT-COUNT <- - SHIFT-COUNT;
         END;
    ELSE SHIFT-DIRECTION <- 0;
         END IF;
    IF SHIFT-COUNT > 64
         THEN SHIFT-COUNT <- 64;
         END IF;
SHIFT-PREP:         END ;


''*       3.2.4.5   DOUBLE LENGTH SHIFT PROCESSOR
***
***                 DOUBLE-SHIFT PERFORMS THE FTSC LONG SHIFTS
***
***       3.2.4.5.1 INPUTS -- SHIFT-DIRECTION IS 1 FOR LEFT SHIFTS.
***                           SHIFT-COUNT IS THE SHIFT LENGTH.
***                           REG-OP IS THE CONTENTS OF THE RB REGISTER
***                           EX IS THE CONTENTS OF THE EXTENSION REGISTER
***       3.2.4.5.2 PROCESS-- SHIFTS LONGER THAN 32 BITS ARE PROCESSED
***                           IN TWO PASSES, 32 BITS AND THE REMAINDER.
***                           A MASK WORD IS FORMED OF LENGTH N, N =
***                           THE SHIFT LENGTH FOR THIS PASS.  LEFT
***                           AND RIGHT BRANCHES ARE CONSIDERED
***                           SEPARATELY.
***                           THE BITS SHIFTED OUT ARE SAVED IN WORK1.
***                           A LOGICAL SHIFT IS PERFORMED ON REG-OP,EX.
***                           A BRANCHED BASED ON THE OP-CODE IS TAKEN
***                           TO FIX UP THE RESULT FOR DIFFERENT SHIFTS.
***       3.2.4.5.3 OUTPUTS-- REG-OP,EX ARE THE RESULT.
***                           SHIFT-COUNT, PASS-COUNT ARE USED AND ZEROED
***                           OVERFLOW MAY BE SET FOR ARITHMETIC LEFT
***                                    SHIFTS                         ''
```

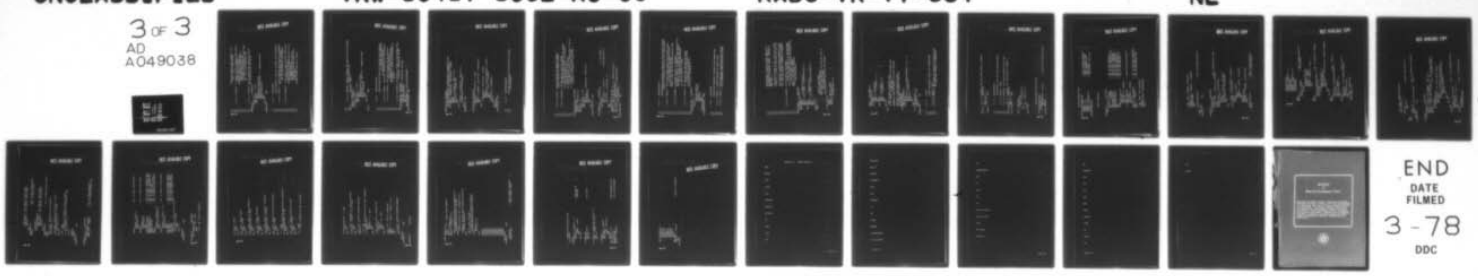
```
DOUBLE-SHIFT:       PROCESSOR;
    SHIFT-PREP;
    IF SHIFT-COUNT > 32
         THEN PASS-COUNT <- 32 ;
         ELSE PASS-COUNT <- SHIFT-COUNT ;
         END IF;
    DO UNTIL SHIFT-COUNT = 0  ;
    BEGIN ;
         SHIFT-MASK33 <- SRA( O'40000000000', PASS-COUNT ) ;
         IF SHIFT-DIRECTION = 0
         THEN BEGIN ;
              WORK1 <- SLL( EX, 32 - PASS-COUNT) AND SHIFT-MASK;
              EX <-     SRL(EX, PASS-COUNT) OR
                        ( SLL (REG-OP, 32 - PASS-COUNT)
                             AND SHIFT-MASK);
              REG-OP <- SRL(REG-OP, PASS-COUNT);
              CASE OPCODE <2:1> ;
                   NULL ;
                   NULL ;
ARL-RS:            BEGIN ;
                        IF SLL(REG-OP,PASS-COUNT) < 0
                             THEN REG-OP <- REG-OP OR SHIFT-MASK ;
                             END IF ;
ARL-RS:                 END ;
                        REG-OP <- REG-OP OR WORK1;
              END CASE ;
         END ;
         ELSE BEGIN ;
                   WORK1 <- REG-OP AND     SHIFT-MASK;
                   REG-OP <- SLL(REG-OP,PASS-COUNT) OR
                        SRL( EX, 32 - PASS-COUNT) ;
                   EX <- SLL (EX, PASS-COUNT) ;
                   CASE OPCODE <2:1> ;
                        NULL ;
                        NULL ;
ARL-LS:                 BEGIN ;
                             IF REG-OP.SIGN-BIT
                                  THEN WORK1 <- WORK1 - SHIFT-MASK;
```

```
                                    END IF ;
                                IF WORK1 /= 0
                                    THEN SET-OVERFLOW ;
                                    END IF ;
ARL-LS:                         END ;
                                EX <- EX OR SRL( WORK1, 32 - PASS-COUNT );
                            END CASE ;
                        END ;
                    END IF ;
                    SHIFT-COUNT <- SHIFT-COUNT - PASS-COUNT ;
                    PASS-COUNT <- SHIFT-COUNT ;
                END ;
DOUBLE-SHIFT:       END;


''*     3.2.5.1   FLOATING-PREP
***
***               FLOATING-PREP SEPARATES REG-OP AND OPERAND
***               INTO MANTISSA AND EXPONENT WORDS.
***
***     3.2.5.1.1 INPUTS -- REG-OP AND OPERAND
***
***     3.2.5.1.2 PROCESS-- THE SIGN-EXTENDED RIGHT-JUSTIFIED
***                         MANTISSA AND EXPONENT PARTS OF BOTH
***                         INPUTS ARE FORMED.
***
***     3.2.5.1.3 OUTPUTS-- OPERAND-EXP AND REG-OP-EXP CONTAIN  THE EXPONENTS
***                         OPERAND AND REG-OP CONTAIN THE MANTISSAS ''
FLOATING-PREP:      PROCESSOR;
    OPERAND-EXP <- SE (OPERAND.EXPONENT);
    REG-OP-EXP  <- SE ( REG-OP.EXPONENT);
    OPERAND     <- SE (OPERAND.MANTISSA);
    REG-OP      <- SE ( REG-OP.MANTISSA);
FLOATING-PREP:      END;
```

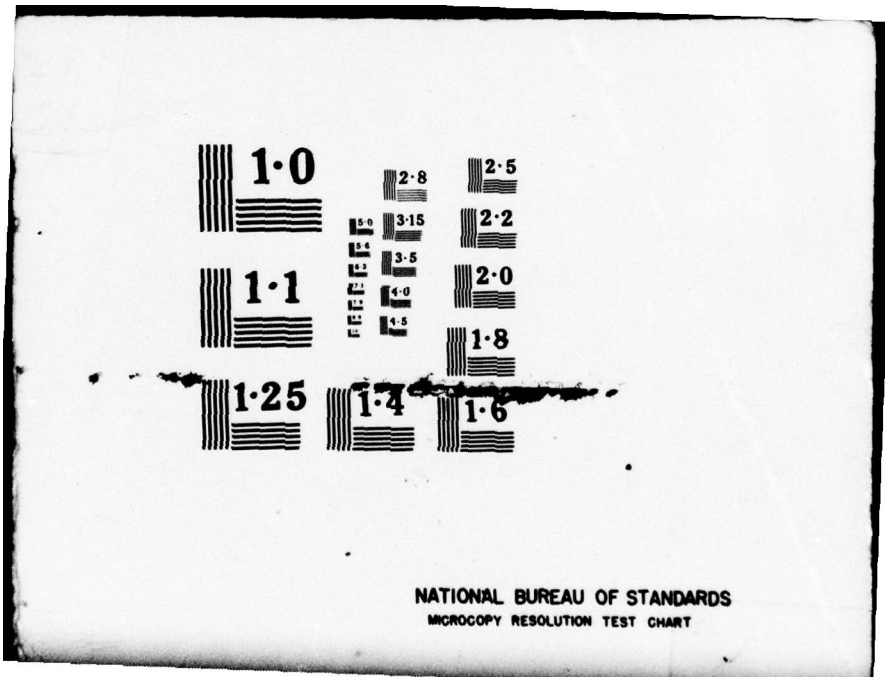
```
''*       3.2.5.2   NORMALIZE-SHIFT (VALUE)
***
***                 NORMALIZE-SHIFT PERFORMS A 32 BIT NORMALIZATION
***
***       3.2.5.2.1 INPUTS -- THE CALLING PARAMETER SPECIFIES THE
***                           VARIABLE TO BE NORMALIZED.
***
***       3.2.5.2.2 PROCESS-- THE VARIABLE IS SHIFTED ITERATIVELY
***                               UNTIL THE MOST SIGNIFICANT DATA BIT
***                               DIFFERS FROM THE SIGN BIT.
***                           THE CALLING ROUTINE MUST PROTECT AGAINST
***                               AN ALL-ZERO VALUE.
***
***       3.2.5.2.3 OUTPUTS-- THE NORMALIZED DATA WORD IS STORED BACK
***                               INTO THE CALLING PARAMETER WORD.
***                           NORMALIZE-COUNT IS THE NUMBER OF SHIFTS
***                               REQUIRED TO NORMALIZE THE VALUE
***                           NORM-SIGN IS THE SIGN OF THE VALUE       ''
NORMALIZE-SHIFT:   PROCESSOR( VALUE );
    DECLARE VALUE                              ;
    NORMALIZE-COUNT <- 0 ;
    NORM-SIGN <- VALUE.SIGN-BIT ;
    DO WHILE VALUE.BIT-30 = NORM-SIGN ;
        BEGIN;
            NORMALIZE-COUNT <- NORMALIZE-COUNT + 1 ;
            VALUE <- SLL(VALUE,1) ;
        END;
NORMALIZE-SHIFT:   END;


''*       3.2.5.3   NORMALIZE
***
***                 NORMALIZE PERFORMS A NORMALIZATION OF THE FLOATING
***                     POINT WORD IN REG-OP.
***
***       3.2.5.3.1 INPUTS -- REG-OP IS THE FLOAT POINT NUMBER TO
***                               NORMALIZE.  THE FORMAT OF THIS WORD
***                               MUST BE A SIGN-EXTENDED RIGHT-JUSTIFIED
***                               WORD.
```

AD-A049 038 TRW DEFENSE AND SPACE SYSTEMS GROUP REDONDO BEACH CALIF F/G 9/2
SMITE REFERENCE MANUAL.(U)
NOV 77
UNCLASSIFIED TRW-30417-6002-RU-00 RADC-TR-77-364 F30602-77-C-0089 NL

3 OF 3
AD
A049038



END
DATE
FILMED
3-78
DDC


1·0
1·1
1·25
1·4
1·6
1·8
2·0
2·2
2·5
2·8
3·15
3·5
4·0
4·5
5·0
NATIONAL BUREAU OF STANDARDS
MICROCOPY RESOLUTION TEST CHART

```
***                          REG-OP-EXP CONTAINS THE EXPONENT OF THE
***                              RESULT BEFORE NORMALIZATION.
***
***      3.2.5.3.2 PROCESS-- IF REG-OP IS ZERO, FLOATING POINT ZERO IS
***                              STORED BACK INTO REG-OP
***                          OTHERWISE, REG-OP IS NORMALIZED.
***                              THEN THE NORMALIZATION COUNT IS
***                              SUBTRACTED FROM REG-OP-EXP.
***                              8 IS ADDED TO REG-OP-EXP, SINCE THERE
***                              ARE 7 OR 8 SIGN EXTENSION BITS
***
***      3.2.5.3.3 OUTPUTS-- REG-OP CONTAINS THE NORMALIZED RESULT.
***                          THE OVERFLOW SIGNAL MAY BE SET              ''
NORMALIZE:           PROCESSOR;
    IF REG-OP = 0
         THEN REG-OP <- X'80' ;
         ELSE BEGIN;
              NORMALIZE-SHIFT(REG-OP);
              REG-OP-EXP <- REG-OP-EXP + 8 - NORMALIZE-COUNT;
              TEST-FP-OVERFLOW ;
              END;
         END IF;
NORMALIZE:           END;


''*      3.2.5.4   FLOAT(BASE)
***
***                FLOAT PERFORMS A FIXED TO FLOATING POINT CONVERSION
***
***      3.2.5.4.1 INPUTS -- REG-OP CONTAINS THE VALUE TO BE CONVERTED
***                          THE CALLING PARAMETER BASE CONTAINS
***                              THE INITIAL OFFSET OF THE EXPONENT
***
***      3.2.5.4.2 PROCESS-- IF THE VALUE IS ZERO, FLOATING POINT ZERO
***                              IS STORED IN REG-OP.
***                          OTHERWISE, REG-OP IS NORMALIZED BY A CALL
***                              TO NORMALIZE-SHIFT.
***                              THE NORMALIZING SHIFT COUNT IS
***                              SUBTRACTED FROM THE LOWER 8 BITS
***                              OF THE BASE.
***                          FLOATING POINT OVERFLOW IS TESTED
```

```
***
***       3.2.5.4.3 OUTPUTS-- REG-OP CONTAINS THE RESULT OF THE CONVERSION
***                           OVERFLOW MAY BE SET                          ''
FLOAT:              PROCESSOR( IBASE );
    DECLARE IBASE<17:0> ;
    REG-OP <- OPERAND ;
    IF REG-OP = 0
        THEN REG-OP <- X'80' ;
        ELSE BEGIN ;
             NORMALIZE-SHIFT( REG-OP);
             REG-OP-EXP <- SE( IBASE<7:0>) - NORMALIZE-COUNT ;
             TEST-FP-OVERFLOW;
             END;
        END IF;
FLOAT:              END;



''*       3.2.5.6   TEST-FP-OVERFLOW
***
***                 TEST-FP-OVERFLOW TESTS FOR FLOATING POINT OVERFLOW
***                                      OR  FLOATING POINT UNDERFLOW
***                               IT ALSO STORES THE EXPONENT.
***       3.2.5.6.1 INPUTS -- REG-OP-EXP CONTAINS THE 18 BIT EXPONENT.
***
***       3.2.5.6.2 PROCESS-- THE CARRY AND SIGN BITS ARE ANALYZED
***                           FOR AN OVERFLOW CONDITION.
***                         SET-OVERFLOW IS CALLED FOR OVERFLOWS.
***                         THE EXPONENT IS STORED AS THE EXPONENT OF
***                           REG-OP
***
***       3.2.5.6.3 OUTPUTS-- REG-OP-EXP IS STORED IN THE EXPONENT PART
***                           OF REG-OP.  OVERFLOW MAY BE SET.          ''
TEST-FP-OVERFLOW:  PROCESSOR;
    IF REG-OP-EXP. EXP-CARRY /= REG-OP-EXP. EXP-SIGN
        THEN SET-OVERFLOW;
        END IF;
    REG-OP.EXPONENT <- REG-OP-EXP.EXPONENT ;
TEST-FP-OVERFLOW:  END;
```

```
UNNORMALIZE: PROCESSOR;
     ''ADJUST EXPONENTS OF FLOATING-POINT OPERANDS SO THAT
     THEY MATCH, AND SHIFT THE MANTISSAS ACCORDINGLY.''
     IF (WORK1 <- OPERAND-EXP - REG-OP-EXP) > 0 THEN
          BEGIN;
          REG-OP <- SRA(REG-OP, WORK1<17:0>);
          REG-OP-EXP <- OPERAND-EXP;
          END;
     ELSE
          OPERAND <- SRA(OPERAND, - WORK1<17:0>)
          ''OPERAND-EXP IS NOT NEEDED'';
     END IF;
UNNORMALIZE: END;



FLOATING-ADD: PROCESSOR;
     IF OPERAND-EXP > REG-OP-EXP + 23 THEN
          BEGIN;
          REG-OP <- OPERAND;  REG-OP-EXP <- OPERAND-EXP;
          END;
     ELSE
          IF OPERAND-EXP > REG-OP-EXP - 23 THEN
               BEGIN;
               UNNORMALIZE  ''ADJUST EXPONENTS TO MATCH'';
               REG-OP <- REG-OP + OPERAND;
               END;
          ''ELSE REG-OP IS UNCHANGED.''
          END IF;
     END IF;
     NORMALIZE;
FLOATING-ADD: END;



'''*      3.2.6.2   FLOATING-MULTIPLY
***
***                 FLOATING MULTIPLY MULTIPLIES TWO FTSC FLOATING
***                      POINT OPERANDS.
```

```
***
***       3.2.6.2.1 INPUTS -- REG-OP CONTAINS THE REGISTER OPERAND.
***                           OPERAND CONTAINS THE MEMORY OPERAND
***
***       3.2.6.2.2 PROCESS-- FLOATING-PREP IS CALLED TO EXTRACT
***                                THE MANTISSA AND EXPONENT OP EACH OPERAND
***                           REG-OP IS LEFT SHIFTED 8 BITS (THIS REMOVES
***                                NON-SIGNIFICANT BITS)
***                           AN INTEGER MULTIPLY IS PERFORMED
***                                THE PRODUCT OF REG-OP * OPERAND GOES TO REG-OP
***                           IF THE RESULT IS ZERO, FP ZERO IS STORED
***                                OTHERWISE, DBL-NORMALIZE (REG-OP,EX)''
FLOATING-MULTIPLY: PROCESSOR                                    ;
    FLOATING-PREP;
    REG-OP <- SLL(REG-OP,8);
    MULTIPLY;
    IF REG-OP = 0
         THEN REG-OP <- 128                                     ;
         ELSE BEGIN                                             ;
              REG-OP-EXP  <- OPERAND-EXP + REG-OP-EXP;
              DBL-NORMALIZE;
              END;
         END IF ;
FLOATING-MULTIPLY: END                                          ;


DBL-NORMALIZE:      PROCESSOR;
    IF SRL( REG-OP + 1, 1) = 0                                  '' -1 OR 0 PASS  ''
         THEN BEGIN;
              REG-OP<30:0> <- EX<31:1> ;                        '' KEEP SIGN,MOVE EX ''
              EX <- 0 ;
              REG-OP-EXP <- REG-OP-EXP -31;
         END;
         END IF;
    NORMALIZE-SHIFT (REG-OP) ;
    REG-OP <- REG-OP OR SRL( EX, 32 - NORMALIZE-COUNT ) ;
    EX <- SLL( EX, NORMALIZE-COUNT);
    REG-OP-EXP <- REG-OP-EXP + 8 - NORMALIZE-COUNT ;
              TEST-FP-OVERFLOW;
DBL-NORMALIZE:      END;
```

```
'!*       3.2.6.3   VECTOR-MULTIPLY
***
***                 VECTOR-MULTIPLY PERFORMS A THREE WORD VECTOR MULTIPLICATION
***                       BETWEEN THE REGISTER VECTOR AND THE OPERAND VECTOR.
***
***       3.2.6.3.1 INPUTS -- REG-OP AND OPERAND CONTAIN THE INITIAL VALUES
***                             FOR THE REGISTER AND OPERAND VECTORS.
***                           EFFAD IS THE EFFECTIVE ADDRESS
***                           AMZERO IS TRUE FOR THE REGISTER-REGISTER MODE
***
***       3.2.6.3.2 PROCESS-- A FLOATING MULTIPLY IS PERFORMED
***                           THE RESULT (REG-OP) IS STORED IN THE REGISTER FILE
***                           LOAD-NEXT IS CALLED TO READ THE NEXT VALUES
***                                OF THE REGISTER AND OPERAND VECTOR.
***                           THE LOOP IS EXECUTED 3 TIMES.
***
***       3.2.6.3.3 OUTPUTS-- THE REGISTER FILE CONTAINS THE RESULT
***                                OF THE FIRST TWO MULTIPLICATIONS.
***                           THE THIRD RESULT IS IN REG-OP.
***                           RB HAS BEEN INCREMENTED BY 2 ''
VECTOR-MULTIPLY:   PROCESSOR                              ;
    DO FOR 1 TO 2   ;
        BEGIN ;
            FLOATING-MULTIPLY;
            GPXR[RB] <- REG-OP;
            LOAD-NEXT;
            END;
    FLOATING-MULTIPLY;
VECTOR-MULTIPLY:   END                                              ;


'!*       3.2.6.4   SQUARE-ROOT
***
***                 SQUARE-ROOT PERFORMS A FLOATING POINT SQUARE ROOT.
***
***       3.2.6.4.1 INPUTS -- OPERAND CONTAINS THE DATA TO USE.
***
***       3.2.6.4.2 PROCESS-- THIS SQUARE ROOT ALGORITHM IS VERY SIMILAR
***                           TO THE ELEMENTARY SCHOOL METHOD OF FINDING
```

```
***                          SQUARE ROOTS. A RULE OF 20 WAS USED TO AID
***                          IN CALCULATING THE NEXT DIGIT ( PARTIAL
***                          REMAINDER / 20*PARTIAL ANSWER = ESTIMATE OF
***                          NEXT DIGIT).
***
***                          OPERATING ON BINARY DIGITS (2 BITS AT A TIME)
***                          A RULE OF 2 IS USED TO CALCULATE THE NEXT BIT.
***
***                          THE ALGORITHM IS COMPLICATED SOMEWHAT BY THE
***                          FACT THAT THE PARTIAL REMAINDER IS CALCULATED
***                          IN A LOOK-AHEAD MANNER, ASSUMING AN ANSWER=1.
***
***      3.2.6.4.3 OUTPUTS-- REG-OP CONTAINS THE RESULT.
***                               DURING THE ALGORITHM IT IS THE REMAINDER
***                          WORK1 IS THE REMAINING VALUE OF OPERAND AFTER
***                               EACH STEP WHEN 2 MORE BITS ARE SHIFTED OUT.
***                          WORK2 IS THE PARTIAL ANSWER.
***                          WORK3 IS THE PARTIAL REMAINDER
***                               (WITH LOOK AHEAD). ''
SQUARE-ROOT:        PROCESSOR                                        ;
    IF OPERAND.LOW-BIT = 0
    THEN BEGIN                                ;
         REG-OP     <- OPERAND.BITS-30-29      ;
         WORK1 <- SLL( WORK1 <- OPERAND.MANTISSA,11 );
         END                                  ;
    ELSE BEGIN                                ;
         REG-OP     <- OPERAND.BIT-30          ;
         WORK1 <- SLL( WORK1 <- OPERAND.MANTISSA,10 );
         END                                  ;
         END IF ;
    WORK2 <- 0 ;
    WORK3  <- 1                               ;
                  ''LOOP FOR THE ANSWER
                  FIND THE PARTIAL REMAINDER
                  SHIFT OVER 2 BITS AND OR IN THE NEXT TWO BITS OF
                       THE ORIGINAL OPERAND
                   SHIFT THOSE TWO BITS OUT OF THE OPERAND  ''
    DO FOR 1 TO 23 ;
         BEGIN;
         WORK3 <- REG-OP - WORK3              ;
         REG-OP<- SLL( WORK3,2)
                  OR WORK1.BITS-31-30         ;
```

```
          WORK1  <-SLL( WORK1,2)                    ;
          IF WORK3.SIGN-BIT
          THEN BEGIN                                ;
               WORK2 <- SLL(WORK2,1)                     ;
               WORK3 <- -( SLL(WORK2,2) + 3) ;
               END                                       ;
          ELSE BEGIN                                    ;
               WORK2 <- SLL(WORK2,1) + 1                 ;
               WORK3 <- SLL(WORK2,2) + 1                 ;
               END                                            ;
          END IF;
     END ;
          ''TEST FOR ZERO OR NEGATIVE HERE
          IF THE OPERAND IS NEGATIVE, SET THE OVERFLOW AND
                         DO NOT STORE
          OTHERWISE STORE THE RESULT IN THE FTSC REGISTER
*
          THEN TEST IF THE OPERAND WAS ZERO
          IF IT WAS, STORE FLOATING POINT ZERO
          IN THE RESULT     ''
     IF OPERAND.SIGN-BIT
          THEN SET-OVERFLOW ;
          ELSE BEGIN ;
               REG-OP-EXP <- SRA( OPERAND.EXPONENT + 1,1 ) - 8;
               REG-OP <- SLL(WORK2,8) ;
               NORMALIZE ;
               END ;
          END IF                                    ;
     IF OPERAND.MANTISSA = 0
          THEN REG-OP <- 128                        ;
          END IF                                    ;
SQUARE-ROOT:        END                                  ;


''*       3.2.7.1   ILLEGAL
***
***                 ILLEGAL PROCESSES ILLEGAL FTSC INSTRUCTIONS.
***
***       3.2.7.1.1 INPUTS -- NONE
***       3.2.7.1.2 PROCESS-- THE ILLEGAL OP CODE BIT IN STATUS IS SET
***                           THE POWER DOWN INTERRUPT IS REQUESTED.
```

```
***       3.2.7.1.3 OUTPUTS-- THE FTSC STATUS WORD IS ALTERED ''
ILLEGAL:            PROCESSOR;
          ILLOP <- 1;
          POWER-REQ <- 1;
ILLEGAL:            END;


''*       3.2.7.2   SET-OVERFLOW
***
***                 SET-OVERFLOW SETS THE FTSC OVERFLOW STATUS
***
***       3.2.7.2.1 INPUTS -- NONE
***
***       3.2.7.2.2 PROCESS-- THE OVERFLOW STATUS BIT IS SET.
***
***       3.2.7.2.3 OUTPUTS-- THE STATUS WORD IS ALTERED ''
SET-OVERFLOW:       PROCESSOR;
          OVERFLOW <- 1;
SET-OVERFLOW:       END;


JUMP:  PROCESSOR;
     IF AMZERO THEN
          PC <- OPERAND<15:0>;
     ELSE
          PC <- EFFAD<15:0>;
     END IF;
JUMP:  END;


''        DEFINE INITIAL MACHINE STATE                              ''

INITIALIZE:PROCESSOR;
    IF MEMTYPE = 0
          THEN MEMTYPE <- X'F0FF' ;      '' IF NOT SET BY USER-ALL WRITE''
          END IF;
INITIALIZE:END;
```

```
''        MACHINE MASTER CLEAR                                          ''

MASTER-CLEAR:PROCESSOR;
     INTREQ <- 0;                          '' CLEAR INTERRUPT REQUESTS   ''
     INPROC <- 0;                          '' CLEAR IN PROCESS FLAGS     ''
     INTMASK <- 0;                         '' CLEAR INTERRUPT MASK       ''
     STATUS <- B'00000111';                '' CLEAR STATUS               ''
     EXECUTE <- 0;                         '' CLEAR EXECUTE INST FLAG    ''
MASTER-CLEAR:END;


''        BEGIN EMULATION                                               ''


OPCODE-0X:          PROCESSOR;
    CASE OPCODE<3:0>;
         REG-OP <- OPERAND;              '' LDR  00 LOAD REGISTER ''
         EX     <- OPERAND;              '' LDE  01 LOAD EXTENSION REG ''
         FETCH-MULTIPLE(2);              '' LDR2 02 LOAD 2 REGISTERS  ''
         FETCH-MULTIPLE(3);              '' LDR3 03 LOAD 3 REGISTERS  ''
         FETCH-MULTIPLE(7);              '' LDR7 04 LOAD 4 REGISTERS  ''
         LDN;                            '' LDN  05 LOAD NEGATIVE ''
         LDNF;                           '' LDNF 06 LOAD NEG FLOATING ''
         IF OPERAND.SIGN-BIT             '' LDA  07 LOAD ABSOLUTE ''
              THEN LDN;
              ELSE REG-OP <- OPERAND ;
              END IF;
         IF OPERAND.SIGN-BIT             '' LDAF 08 LOAD ABS FLOATING ''
              THEN LDNF;
              ELSE REG-OP <- OPERAND;
              END IF ;
         REG-OP <- NOT OPERAND;          '' LDC  09 LOAD ONES COMPLEMENT''
         REG-OP <- OPERAND;              '' LAO  0A LOAD ACTIVE ONLY ''
         NULL;                           '' LMO  0B LOAD MONITOR ONLY ''

          ''0C - ADDF - ADD (FLOATING)''
          BEGIN;
          FLOATING-PREP;  FLOATING-ADD;
          END;
```

```
          ''0D - SUBF - SUBTRACT (FLOATING)''
          BEGIN;
          FLOATING-PREP;  OPERAND <- - OPERAND;
          FLOATING-ADD;
          END;

        FLOATING-MULTIPLY;              '' MPYF 0E FLOATING MULTIPLY ''
  NULL;
    END CASE;
OPCODE-0X:          END;



OPCODE-1X:          PROCESSOR;
    CASE OPCODE<3:0>;
        SQUARE-ROOT ;                   '' SRTF 10 SQUARE ROOT FLOAT   ''
         ''11 - VADDF - VECTOR ADD (FLOATING)''
         BEGIN;
         FLOATING-PREP;  FLOATING-ADD;
         DO FOR 1 UP TO 2;
              BEGIN;
              GPXR[RB] <- REG-OP;  LOAD-NEXT;
              FLOATING-PREP;  FLOATING-ADD;
              END;
         END;

         ''\\ - VSUBF - VECTOR SUBTRACT (FLOATING)''
         DO FOR WORK2 <- 0 UP TO 2;
              BEGIN;
              IF WORK2 /= 0 THEN
                   BEGIN;
                   GPXR[RB] <- REG-OP;  LOAD-NEXT;
                   END;
              END IF;
              FLOATING-PREP;
              OPERAND <- - OPERAND;  FLOATING-ADD;
              END;

        VECTOR-MULTIPLY;                '' VMPYF 13 VECTOR MULTIPLY ''
        BEGIN;                          '' VIPF 14 INNER PRODUCT ''
```

```
          VECTOR-MULTIPLY                                    ;
          OPERAND <- GPXR[RB - 1];
          FLOATING-PREP;
          FLOATING-ADD                                       ;
          OPERAND <- GPXR[RB - 2];
          FLOATING-PREP;
          FLOATING-ADD                                       ;
END;
BEGIN;                              '' VSMF 15 SCALAR MULTIPLY ''
     WORK3 <- OPERAND;              '' SAVE OPERAND ''
     DO FOR 1 TO 3;
          BEGIN;
               FLOATING-MULTIPLY;
               GPXR[RB] <- REG-OP;
               OPERAND <- WORK3;
               REG-OP <- GPXR[ RB <- RB+1 ] ;
               END;
END;
BEGIN;                              '' CFX  16 CONVERT TO INTEGER ''
     IF SE ( OPERAND.EXPONENT) <= 0
     THEN REG-OP <- 0;
     ELSE BEGIN ;
          IF OPERAND.EXPONENT > 31
               THEN SET-OVERFLOW;
               ELSE REG-OP <- SRA(REG-OP,31- OPERAND.EXPONENT);
               END IF;
          END;
     END IF;
END;
BEGIN;                              '' UPF  17 UNPACK ''
     GPXR[RB] <- SE(OPERAND.EXPONENT) ;
     RB <- RB+1;
     REG-OP <- OPERAND ;
     REG-OP.EXPONENT <- 0;
     END;
BEGIN;                              '' PKF  18 PACK ''
     REG-OP-EXP <- SE( OPERAND.EXPONENT);
     LOAD-NEXT;
     FLOAT( REG-OP-EXP);
     END;
 ''\\ - ADD - INTEGER ADD''
 ADD(OPERAND, 0);
```

```
          ''\\ - SUB - INTEGER SUBTRACT''
          ADD(-OPERAND, 0);

          BEGIN ;
               MULTIPLY;                    '' MULTIPLY AND ADJUST EX ''
               END;
 NULL;
 NULL;
          FLOAT(31);                        '' CFL  1E CONVERT TO FLOATING ''
          REG-OP <- REG-OP AND OPERAND; '' AND 1F ''
     END CASE;
OPCODE-1X:          END;



OPCODE-2X:          PROCESSOR;
     CASE OPCODE<3:0>;
          REG-OP <- REG-OP XOR OPERAND; '' XOR 20 ''
          REG-OP <- REG-OP  OR OPERAND; '' OR   21 ''
          REG-OP <- REG-OP AND NOT OPERAND; '' ANI  22 AND INVERTED ''
          BEGIN;                            '' ARS  23 ARITHMETIC SHIFT ''
               SHIFT-PREP;
               IF SHIFT-DIRECTION
                  THEN REG-OP <- SRA (REG-OP,SHIFT-COUNT) ;
                  ELSE BEGIN;
                       REG-OP <- SLA(REG-OP, SHIFT-COUNT);
                       IF GPXR[RB] /= SRA (REG-OP,SHIFT-COUNT)
                            THEN SET-OVERFLOW;
                            END IF;
                       END;
                  END IF;
               END;
          BEGIN;                            '' ARL  24 DBL ARITHMETIC SHIFT''
               EX-SIGN <- EX.SIGN-BIT;
               DOUBLE-SHIFT;
               EX <- EX-SIGN // EX<31:1> ;
               END;
          BEGIN;                            '' RRS  25 CIRCULAR SHIFT ''
               SHIFT-PREP;
               IF SHIFT-DIRECTION
```

```
                    THEN REG-OP <- SRC (REG-OP, SHIFT-COUNT);
                    ELSE REG-OP <- SLC (REG-OP, SHIFT-COUNT);
                    END IF;
               END;
          DOUBLE-SHIFT;                    '' RRL  26 DBL CIRCULAR SHIFT ''
          BEGIN;                           '' LSS  27 LOGICAL SHIFT ''

               SHIFT-PREP;
               IF SHIFT-DIRECTION
                    THEN REG-OP <- SRL (REG-OP, SHIFT-COUNT);
                    ELSE REG-OP <- SLL (REG-OP, SHIFT-COUNT);
                    END IF;
               END;
          DOUBLE-SHIFT ;                   '' LLS  28 DOUBLE LOGICAL SHIFT''
          NULL;
          NULL;
           ''2B - ACO - INTEGER ADD WITH CARRYOUT''
           ADD(OPERAND, 1);

           ''2C - ASZ - LOGICAL AND AND SKIP IF ZERO''
           IF (OPERAND AND REG-OP) = 0 THEN
                PC <- PC + 1;
           END IF;

           ''2D - OISO
                - LOGICAL OR INVERTED AND SKIP IF ONES''
           IF ((NOT OPERAND) OR REG-OP) = X'FFFFFFFF' THEN
                PC <- PC + 1;
           END IF;

          ILLEGAL;
          ILLEGAL;
     END CASE;
OPCODE-2X:          END;



OPCODE-4X:          PROCESSOR;
     CASE OPCODE<3:0>;
          STORE (REG-OP,EFFAD);            '' STR  40 STORE REGISTER ''
          STORE (EX,    EFFAD);            '' STE  41 STORE EXTENSION REG ''
```

```
        BEGIN;                              '' STD  42 STORE REGISTER DBL ''
             OPERAND <- REG-OP;
             STORE-DOUBLE;
             END;
        BEGIN;                              '' SZD  43 STORE ZERO DOUBLE ''
             OPERAND <- 0;
             STORE-DOUBLE;
             END;
        STORE-MULTIPLE(2);                  '' STD2 44 STORE 2 REGISTER DBL''
        STORE-MULTIPLE(3);                  '' STD3 45 STORE 3 REGISTER DBL''
        STORE-MULTIPLE(7);                  '' STD7 46 STORE 7 REGISTER DBL''
        STORE(0,EFFAD);                     '' SZD  47 STORE ZERO ''
        NULL;                               '' STH  48 STORE HARD ADDRESS ''
        BEGIN;                              '' SPS  49 STORE PC,STAT SINGLE''
             OPERAND <- STATUS // PC;
             STORE (OPERAND,EFFAD);
             END ;
        BEGIN;                              '' SPC  4A STORE PC,STAT DOUBLE''
             OPERAND <- STATUS // PC;
             STORE-DOUBLE;
             END ;
        BAD-STORE;                          '' SBAO 4B BADD ADDRESS PARITY ''
        BAD-STORE;                          '' SBAZ 4C BADD ADDRESS PARITY ''
        BAD-STORE;                          '' SBDO 4D BADD ADDRESS PARITY ''
        BAD-STORE;                          '' SBDZ 4E BADD ADDRESS PARITY ''
         ''4F - JPZ - JUMP IF PLUS OR ZERO''
         IF REG-OP >= 0 THEN
            JUMP;
         END IF;

    END CASE;
OPCODE-4X:          END;


OPCODE-5X:          PROCESSOR;
    CASE OPCODE<3:0>;
          ''50 - JMP - JUMP''
          JUMP;

          ''51 - JMI - JUMP IF NEGATIVE''
```

```
IF REG-OP < 0 THEN
     JUMP;
END IF;

''52 - JZE - JUMP IF ZERO''
IF REG-OP = 0 THEN
     JUMP;
END IF;

''53 - JZEF - JUMP IF ZERO (FLOATING)''
IF REG-OP.MANTISSA = 0 THEN
     JUMP;
END IF;

''54 - JNZ - JUMP IF NON-ZERO''
IF REG-OP /= 0 THEN
     JUMP;
END IF;

''55 - JNZF - JUMP IF NON-ZERO (FLOATING)''
IF REG-OP.MANTISSA /= 0 THEN
     JUMP;
END IF;

''56 - JPS - JUMP IF POSITIVE AND NON-ZERO''
IF REG-OP > 0 THEN
     JUMP;
END IF;

''57 - JPSF
     - JUMP IF POSITIVE AND NON-ZERO (FLOATING)''
IF REG-OP.MANTISSA > 0 THEN
     JUMP;
END IF;

''58 - JMZ - JUMP IF NEGATIVE OR ZERO''
IF REG-OP <= 0 THEN
     JUMP;
END IF;

''59 - JMZF
     - JUMP IF NEGATIVE OR ZERO (FLOATING)''
```

```
            IF REG-OP.MANTISSA <= 0 THEN
                 JUMP;
            END IF;

            ''5A - JDN - DECREMENT RB, JUMP IF NON-ZERO''
            IF (REG-OP <- REG-OP - 1) /= 0 THEN
                 JUMP;
            END IF;

            ''5B - DSI - DISABLE INTERRUPT NETWORK''
            ENI-STAT <- 1;

            ''5C - JOS
                 - JUMP IF OVERFLOW SET AND RESET OVERFLOW''
            IF OVERFLOW THEN
                 BEGIN;
                 OVERFLOW <- 0;  JUMP;
                 END;
            END IF;

            ''5D - JCS - JUMP IS CARRYOUT SET''
            IF CARRY THEN
                 BEGIN;
                 CARRY <- 0;  JUMP;
                 END;
            END IF;

            ''5E - JSB - JUMP TO SUBROUTINE''
            BEGIN;
            INTLEV <- 7;  REG-OP <- STATUS // PC;
            JUMP;
            END;

            ''5F - ENI - ENABLE INTERRUPT NETWORK''
            ENI-STAT <- 0;

        END CASE;
    OPCODE-5X:      END;


    OPCODE-6X:          PROCESSOR;
```

```
    CASE OPCODE<3:0>;
         ''60(\) - RFI - RETURN FROM INTERRUPT''
         BEGIN;
         PC <- OPERAND<15:0>;
         STATUS <- OPERAND<23:16>;

         ''RESET INTERRUPT-IN-PROCESS FLAG SPECIFIED BY
             INTERRUPT-LEVEL FIELD OF OPERAND.''
         INPROC <-
             INPROC AND SLC(X'3FE', OPERAND<18:16>);

         ''(GENERATED CODE FOR ABOVE EXPRESSION SHOULD BE
             CHECKED CAREFULLY.)''
         END;

         ''60(\) - XEC - EXECUTE''
         ''TELL NEXT LOOP OF THE EMULATION CYCLE TO
             EXECUTE #OPERAND# INSTEAD OF MEM[PC]''
         BEGIN;
             EXECUTE <- 1;
             PC <- PC - 1;
         END;

        ILLEGAL;
        ILLEGAL;
        ILLEGAL;
        ILLEGAL;
        ILLEGAL;
        ILLEGAL;
        ILLEGAL;
        ILLEGAL;
        ILLEGAL;
        ILLEGAL;
        ILLEGAL;
        ILLEGAL;
        ILLEGAL;
        ILLEGAL;
        ILLEGAL;
    END CASE;
OPCODE-6X:      END;

     INITIALIZE;                    '' DEFINE MEMORY PROPERTIES   ''
     MASTER-CLEAR;                  '' DEFINE INITIAL STATE       ''
```

```
''        INSTRUCTION EMULATION LOOP                                    ''
 DO FOREVER;

 BEGIN;
IF (ICOUNT <- ICOUNT+1) > 0
     THEN BEGIN;                                  '' TIME FOR ICS       ''
          ICOUNT <- -1000 ;                       '' TEMPORARY          ''
          END;
     END IF;


IF STEP-FLAG
     THEN OP-STEP;                                '' FTSC STEP          ''
     END IF;


IF( WORK1 <- INTREQ AND INTMASK) /= 0
     THEN ''PINSIM'' EX <- 1 ;
     END IF;


IF EXECUTE
     THEN INR <- OPERAND ;
     ELSE BEGIN ;
           INR <- FETCH(PC);
          END ;
     END IF;
PAR1<32:10> <- 0;                                 ''CLEAR EMULATOR FLAG''
PC <- PC + 1;
OPERAND-FETCH;                                    '' RET. EFFAD,OPERAND''
REG-OP <- GPXR[RB] ;
```

```
      CASE OPCODE<6:4> ;
           OPCODE~0X;
           OPCODE~1X;
           OPCODE~2X;
           ILLEGAL;
           OPCODE~4X;
           OPCODE~5X;
           OPCODE~6X;
           ILLEGAL;
      END CASE;
      GPXR[RB] <- REG-OP;                    '' RESULT TO REGISTER''
       END;
FTSC:END;
```

Appendix I. SMITE Keywords

AND

BEGIN

CASE

CLOCK

DATA

DECLARE

DEFAULT

DEFINED

DO

DOWN

ELSE

END

END CASE

END IF

ESCAPE

EXTERNAL

FLAG

FOR

FOREVER

IF

IN

LIGHT

MEMORY

MICROSECONDS

MILLISECONDS

MS

NANOSECONDS

NOT

NS

NULL

OR

OR

PARALLEL-BEGIN

PARALLEL-END

PORT

PROCESSOR

REGISTER

S

SE

SECONDS

SLA

SLC

SLL

SRC

SRL

STEP

SWITCH

THEN

TO

UNTIL

UP

US

WHILE

MISSION
of
Rome Air Development Center

RADC plans and conducts research, exploratory and advanced development programs in command, control, and communications (C^3) activities, and in the C^3 areas of information sciences and intelligence. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.